Automatic Computational Domain Detection

by

Richard Uhrie

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved July 2021 by the
Graduate Supervisory Committee:

John Brunhaver, Chair
Chaitali Chakrabarti
Aviral Shrivastiva
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

August 2021

ABSTRACT

Heterogenous SoCs are in development that marry multiple architectural patterns together. In order for software to be run on such a platform, it must be broken down into its constituent parts, kernels, and scheduled for execution on the hardware. Although this can be done by hand, it would be arduous and time consuming; rather, a tool should be developed that analyzes the source binary, extracts the kernels, schedules the kernels, and optimizes the scheduled kernels for their target component.

This dissertation proposes a decidable kernel definition that enables an algorithmic approach to detecting kernels from arbitrary programs. This definition is built upon four constraints that can be tested using basic graph theory. In addition, two algorithms are proposed that successfully extract kernels based upon runtime information. The first utilizes dynamic traces, which are generated using a collection of novel optimizations. The second utilizes a simple affinity matrix, which has no runtime overhead during program execution. Finally, a Dense Neural Network is proposed that is capable of detecting a kernel's archetype based upon only the composition of the source program and the number of times individual basic blocks execute.

The contributions proposed in this dissertation provide the necessary infrastructure to perform a litany of other optimizations on kernels. By detecting kernels algorithmically, any program can be analyzed and optimized with techniques that have heretofore required kernels be written in a compatible form. Computational kernels can be extracted from any program with no constraints. The innovations describes here will form the foundation for automated kernel optimization in the future, helping optimize the code of the future.

ACKNOWLEDGMENTS

I would like to thank everyone who helped me complete my doctoral education.

First, I would like to thank my advisor, Dr. John Brunhaver. His guidance throughout the process helped me view my work through different lenses, leading to insights I doubt I would have found on my own. His assistance has proven invaluable in inspiring my approach to this work.

Second, I would like to thank Dr. Chaitali Chakrabarti. She was always there to help with my writing and to guide me through the nuances of Academia. She has been an invaluable mentor and has my deepest gratitude.

Third, I would like to thank my father, Dr. John Uhrie. He was always there to help guide me through throughout my education and was never afraid to help edit my works, even though he didn't fully understand them.

I would also like to thank Seth Abraham, Umit Ogras, and Carole Wu for helping edit various papers and providing advise on the course of this research. Thank you to Sean Bryan, Liangliang Chang, Mukul Gupta, Vamsi Lanka, Sriharsha Uppu, and Benjamin Willis for helping write input applications for this work to analyze.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7860. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusion contained herein are those of the authors and should not be interpreted as necessarily representing the social policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Contributions 5

1.2 Future Work 7

2 BACKGROUND 9

2.1 Programs are Directed Graphs of Linear Code 9

2.2 Loops are Kernels 11

2.3 Kernel Transformations 13

2.3.1 DSLs 14

2.3.2 Hardware Accelerators 15

2.4 Synopsis 16

3 AUTOMATED COMPUTATIONAL KERNEL EXTRACTION FROM DYNAMIC APPLICATION TRACES 17

3.1 Introduction 18

3.2 Background and Motivation 21

3.2.1 Current Kernel Segmentation Techniques 22

3.2.2 Demand for Dynamic Kernel Analysis 23

3.3 A Generic Kernel Definition 24

3.3.1 Constraint 1: A Kernel is a Strongly Connected Subgraph 26

3.3.2 Constraint 2: Kernels May Not Partially Overlap 27

3.3.3 Constraint 3: A Kernel Must be More Likely to Recur 28

CHAPTER Page

3.3.4 Implications of These Constraints 29
3.4 Low Overhead Dynamic Traces 30
3.5 Kernel Extraction 33
3.5.1 Initial Kernel Detection 34
3.5.2 Kernel Refinement Algorithms 37
3.6 Results 43
3.6.1 Methodology 43
3.6.2 Dynamic Tracing 45
3.6.3 Kernel Extraction Algorithm 46
3.6.3.1 Extraction Pipeline Kernel Effects 49
3.6.3.2 Hot Code Threshold Effect 51
3.6.3.3 Required Kernel Properties 53
3.6.4 Summary 54
3.7 Discussion 54
3.8 Conclusion 55
3.9 Cross Library Validation 56
4 A REFINED, DECIDABLE KERNEL DEFINITION 59
4.1 Kernel Definition Constraints 60
4.1.1 Constraint 1: A Kernel is a Strongly Connected Subgraph 62
4.1.2 Constraint 2: Every Kernel Must Not Partially Overlap With Any Other Kernel 64
4.2 Constraint 3: A Kernel Must be More Likely to Recur Than to Terminate 66
4.2.1 Constraint 4: The Difference Between Two Nested Kernels Must Contain at Least One Incoming Edge 68

CHAPTER Page

4.2.2 Optimizing Principal 1: Every Block Belonging to a Cycle Must be a Part of a Kernel 71

4.2.3 Optimization Principal 2: Every Kernel is of Minimal Size Such That All Constraints are True 72

4.2.4 Summary of Kernel Properties 72

4.3 Refined Kernel Detection Algorithm 74

4.4 Summary of Updated Kernel Definition 76

5 PREDICTING KERNEL LABELS FROM STATIC AND DYNAMIC PROGRAM INTRINSICS 81

5.1 Introduction 82

5.2 Background 84

5.2.1 Computational Kernel Definitions 85

5.2.2 Extrapolating Program Behavior 88

5.3 Extracting Features from Programs 89

5.3.1 Structure of Features and Kernels 89

5.3.2 Features in Programs 91

5.3.3 Input Programs in Kernel Collection 92

5.3.4 Summary 93

5.4 Domain Reduction 93

5.4.1 Variance Filtering 95

5.4.2 Multivariate Analysis 95

5.4.3 Principal Component Analysis 97

5.4.4 Summary 101

5.5 Kernel Classification 101

CHAPTER Page

5.5.1 Alternative Supervised Classifiers 103

5.5.2 K-Nearest-Neighbor 103

5.5.3 DNN Classifier 105

5.5.4 Library Analysis 109

5.5.5 Summary 111

5.6 Conclusion 111

6 CONCLUSION 113

REFERENCES 117

LIST OF TABLES

Table Page

1. TraceAtlas Test Corpus 7
2. Normalized TraceAtlas Performance Comparison 31
3. TraceAtlas Test Corpus 44
4. Final Cross-Validation Applications 57
5. Descriptions of Kernel Labels 90
6. TraceAtlas Test Corpus 92
7. Top 20 Selected Multivariate Features 97
8. Next 30 Selected Multivariate Features 98
9. Technique Learning Metrics 102
10. DNN Design Performance 105
11. FFT Leave-One-Out Performance 109

LIST OF FIGURES

Figure Page

1. For-Loop CFG 11
2. TraceAtlas Analysis Pipeline 19
3. Kernel Basic Block Diagram with Code 25
4. TraceAtlas Kernel Graph 27
5. Kernel Basic Block Example 30
6. Kernel Detection Analysis Flow 34
7. Kernel Probability Graph 36
8. Basic Block Affinity Calculation 37
9. Basic Block Fusion 38
10. Kernel Smoothing Code 39
11. Trimmed Kernel Structural Flaws 40
12. Trimmed Kernel Pseudocode 40
13. Split Kernel Flaws 41
14. Split Kernel Pseudocode 42
15. Overall Trace Size 46
16. Overall Trace Speed 47
17. Kernel Compliance 48
18. Kernel Count 49
19. Detected Kernels Coverage 50
20. Minimum Hot Code Threshold 52
21. Kernel Entrance and Exit Count 53
22. Constraint 1 Example 63
23. Constraint 1 Pseudocode 63

Figure Page

24. Strongly Connected Subgraphs Overlap 65
25. Constraint 2 Pseudocode 66
26. Probable Path Example 68
27. Constraint 3 Pseudocode 69
28. Constraint 4 Pseudocode 78
29. Kernel Algorithm Pseudocode 79
30. CFG Weight Transformation 80
31. Pure Nested Kernel Example 80
32. TraceAtlas Analysis Pipeline 83
33. Matrix Multiply Kernel Example 86
34. Kernel Labeling Example 90
35. Data Feature Engineering Pipeline 94
36. Multivariate Mutual Information Score 96
37. Accumulated PCA Entropy 99
38. PCA Cosine Similarity 100
39. KNN Accuracy 104
40. DNN Structure 106
41. Multivariate Effect on DNN Accuracy 107
42. Label Prediction Metrics 108

Chapter 1

INTRODUCTION

Computer architects achieve significant energy efficiency through the development of computing accelerators. In thermally constrained environments, while executing embarrassingly parallel workloads, energy efficiency maximizes performance (Power = Energy/op * Op/sec). Accelerators exploit the structure of an algorithm or class of algorithms to reduce the energy cost of executing an operation[1]. For example, many signal processors utilize special registers and memories to capture locality in the computation, reducing memory traffic[2]. Additionally, eliminating the overhead of performance features for scalar compute and amortizing control overheads across identical operations reduces energy costs further[1]. Finally, spatially distributed architectures, like coarse-grained reconfigurable arrays, exchanges communication-in-time through memories for communication-in-space over wires, further reducing energy[3][4]. These techniques rely on properties intrinsic to the computation, though.

This class of algorithms suitable for acceleration is a subset of compute kernels. Here "kernel" as in "central or essential"[5] indicates a primal property of compute kernels, that they have a recurrent calculation component. Collela, while presenting his seven dwarfs[1], emphasized a similarity in arithmetic and memory patterns inside each class[6][7]. Asanovic expanded the number of dwarves to thirteen, looking beyond the initial space of scientific computing to broadly incorporate other application domains. However, Asanovic maintained a similar focus on "similarity in computation and data movement," persistent "underlying patterns," and "equivalence classes"[7].

[1] another name for compute kernels

It stands to reason that additional kernel categories must be added over time to reflect the most important kernels of tomorrow. For example, Asanovic has since updated the list of kernel dwarves from thirteen dwarves targeting HPC to twenty-five broader kernels. As new programming paradigms are developed, it is necessary to develop new kernel categories to succinctly represent the computational landscape. As an example, neural networks were in their infancy just ten years ago. The types of kernels in those algorithms were largely unexplored due to them being as of yet insignificant. Today, neural networks are widespread, with them influencing virtually every field of computation. It is likely that other such advancements will create new kernels that will enter the lexicon. If kernels continue to be defined as a collection of categories, the list of computations defined as kernels will be in constant flux.

To provide a stable interface for developers, Domain Specific Languages (DSLs) have been developed to describe a single class of kernels by embedding the unique properties of the kernel in the language's syntax. For example, halide specializes in image processing kernels and can infer that each kernel will both stride across the input images and that the kernel will be a part of an image processing pipeline[8]. These inferences allow halide to better optimize the code by scheduling both inter and intra kernel optimizations around these properties. When compared to an expert programmer, halide can achieve a speedup of up to 5.9x in up to 18x fewer lines of code. Other DSLs can achieve similar levels of performance be specifically exploiting the nature of their specialized kernels. Similarly, DSLs can also simplify cross compilation of kernels. An FFT, for example, has the potential of being sped up by 75% while simultaneously lowering energy consumption by at least 75% through the design of a simple FFT coprocessor[9]. DSLs allow kernels to be more easily optimized for uses such as these.

Unfortunately, most developers are unable to use these innovations due to the high

barrier of entry. The use of a DSL requires a developer to understand both the intricacies of their code and the nature of the platform they are targeting. This constraint limits the number of developers capable of using such tools to those who are an expert at their craft.

The relative skill, and thus cost, of the developers needed to utilize these tools limits the use of these tools to only the most performance critical code. It is not economical today to employ these techniques at every kernel present in a computation. Due to limited resources, engineers are funneled into working on the tasks with the greatest return on investment. It is easy to justify using a team of engineers to accelerate a kernel that occupies 60% of a server's runtime. It is more difficult to justify it on smaller computations that occupy only 5%. Ultimately, the adoption of a technology is proportional to the cost required to deploy it. Manual processes such as these see less use than more automated techniques.

There is no automated method of identifying and classifying kernels in arbitrary programs today. Cross compilation requires deliberate algorithm design. DSL utilization requires understanding both the DSL and the original source in depth. HLS requires the developer to manually specify optimization parameters. These techniques can produce amazing results if the developers have the time and knowledge to use them. Unfortunately, that is not the world we are living in today. Engineers are a limited commodity, and they can only invest so much time on each project. Automated approaches will help alleviate these problems, enabling far more software to use these techniques.

This dissertation seeks to enable the extraction of kernels from naïve programs and facilitates kernel labelling. Many techniques have been developed to accelerate programs in the HPC space, but they have not seen widespread adoption except for in a few limited scenarios. Providing an automated tool that could apply these techniques would dramatically lower the cost of adoption. This dissertation only provides the tools necessary for the identification and classification of kernels. Subsequent work will be necessary to apply

potential optimizations. The primary motivation of this work is to identify kernels from wild code such that they can be utilized by accelerators without any expert interaction.

The chief contribution this dissertation provides is a decidable kernel definition that can be tested in polynomial time. By marrying the runtime behavior of a program to the static computational graph, it is possible to acquire a holistic understanding of how a program operates. This information allows for testing arbitrary code sequences against a set of constraints. The constraints this work proposes have been designed to conform with prior kernel definitions and be able to be tested quickly to ensure it is practical.

It is theoretically possible to find new classes of kernels, but most if not all the computationally significant kernels today have already been identified. HPC engineers have been optimizing the most significant kernels in server workloads since the inception of computing. It is unlikely that there are many untapped kernels in modern workloads. This dissertation addresses how to find the kernels required by those works.

In the past, more widespread adoption of accelerator technology led to the identification of additional use cases beyond the scope of the original intent. The GPU was originally designed simply to offload graphical computations from the CPU. Once Nvidia provided the CUDA interface, it became a viable compilation target for any vector floating-point program. Today, GPUs are used for countless scientific workloads, from linear algebra to neural networks to image processing. More widespread adoption of the technology allows for more people to utilize the technology, potentially finding new novel methods of utilizing the hardware. Any approach enabling such use is inherently good. Automated kernel extraction and classification is the first step in enabling such processes.

The tools as they exist today primarily guide a developer on the best path forwards. TraceAtlas identifies kernels in the source code, annotates them for the user, and provides a corresponding kernel label. An example could include identifying a function as an FFT.

With this information available, the developer would be guided to using an accelerator or inserting a more optimal FFT library such as FFTW. As an additional example, if a code segment is identified as a stencil operation, it would be advisable for the developer to rewrite that code in a stencil DSL such as halide or in CUDA for hardware acceleration. These are tasks that are easy to do if the developer fully understands their code and the optimization paths, but by providing it even the most inexperienced developers could choose to perform such tasks. This is the advantage of TraceAtlas. One day it will be possible for these optimizations to be applied automatically, but the information provided today is enough to enable even inexperienced developers to take advantage of the bountiful optimizations in the HPC space.

1.1 Contributions

This dissertation's primary contribution is a decidable kernel definition; however, various other contributions were necessary to test this definition and demonstrate its utility. First, This dissertation proposes a combination of dynamic tracing optimizations that lowers the runtime overhead well below current state of the art techniques. With these optimizations, it is possible to trace significantly larger applications. Second, this dissertation provides a decidable kernel definition that allows for arbitrary code segments to be tested for kernel membership. Current techniques require a developer hand-label their kernels, limiting their application to only the most important computations. Third, this dissertation proposes two different kernel extraction algorithms. Fourth and finally, this dissertation provides a machine learning classifier that labels kernels with high fidelity. Prior classifiers were either low-accuracy, limited to two labels, or trained against very small sample sets. The dissertation explains this in the following chapters:

Chapter 3 describes a basic kernel definition and how it is possible to utilize it to extract kernels from dynamic traces. It starts by proposing a kernel definition that builds upon the established CFG representation used in many compilers. Afterwards, the chapter describes the necessary optimizations to dynamic tracing such that it is a viable technique for program analysis. Finally it describes how to analyze the dynamic traces to identify kernels from any program.

Chapter 5 describes a process of applying machine learning to kernels to predict what category they belong to. Traditional ML techniques such as SVM, KNN, and Logistic Regression cannot predict a kernel's archetype with high accuracy. The sample space is non-continuous, requiring a more complex classifier for robust prediction. The chapter concludes by proposing a DNN that is capable of predicting a kernel's archetype with 96.8% accuracy.

Chapter 4 goes into greater depth on the kernel definition and adds additional constraints required to produce complete kernels. There are four constraints that can be applied to a collection of basic blocks to test its membership in the kernel class. In addition, this chapter proposes two optimization principles to ensure that all the kernels present in a program are detected. By using these techniques it is possible to develop an algorithm to extract kernels from any program. The chapter concludes with a proposed algorithm that has been successfully applied to the programs tested in this work.

Chapters 3 and 4 are documents that have been submitted to IEEE Transactions on Computers. Per ASU requirements, the papers appear in this work unedited, save for formatting changes to match the rest of the document. In these chapters there are references to a collection of programs used to evaluate kernels. Over time more programs have been added, leaving the version used in the paper out of date. Table 1 contains the final set of kernels used in this work. Although the figures and discussion in those chapters only show

Table 1: TraceAtlas Test Corpus

Library	Applications	Kernels	Version
StreamIt[10]	16	137	1.6
Halide[8]	41	977	2019/08/27
Gnu Scientific[11]	264	3,157	2.6
Perfect[12]	51	264	1.0.0
FEC[13]	8	167	3.0.1
MiBench[14]	27	170	1.0
SHOC[15]	10	35	1.1.4
VOLK	57	129	2.0.0
FFTW[16]	84	4,686	3.3.8
Dhrystone and Whetstone[17]	7	74	2
OpenCV[18]	74	6,827	4.1.0
mbedTLS[19]	70	1,902	2.16.3
Cortex Suite[20]	19	702	2019/10/30
Eigen[21]	47	2,958	3.3.6
FFmpeg	9	66	4.2
spuce[22]	93	849	0.4.3
LiquidSDR[23]	114	1,409	1.3.1
Armadillo[24]	184	1,496	9.600.6
SigPack	56	2,243	1.2.4
In-House	590	3,015	
Total	1,821	31,263	

a smaller subset of the data, they do extend to the newer programs that have been added as well.

1.2 Future Work

This work does not solve the auto-parallelism problem, but it does provide the required infrastructure to eventually solve it. A holistic kernel definition is necessary to identify what needs to be parallelized. Such a definition is proposed in Chapters 3 and 4. A method of extracting kernels is required for any optimizations to occur, see Chapters 3 and 3 for algorithms that can efficiently identify said kernels. This work does not explore actually

optimizing the kernels identified, but instead provides all the tools required to identify and isolate them.

The first area to expand from this work is to further expand the classes of kernels analyzed by this definitions. The kernels provided are representative of modern radio processing applications. Although it is suspected that the results within this work will extend to all kernel classes, this dissertation can only make claims about the types of applications it analyzes. Further analysis may be required from other domains to verify it fully encompasses all flavors of computational kernels.

The second area to expand is to apply modern optimization techniques to the identified kernels. This could include heterogenous platform scheduling, auto-parallelizing code based upon runtime information, asynchronous kernel scheduling of naive programs, AI assisted program design tools, amongst others. This work lays the foundation for works such as these.

Chapter 2

BACKGROUND

2.1 Programs are Directed Graphs of Linear Code

Compilers utilize an intermediate representation (IR) that abstracts code into a directed graph called a control flow graph (CFG)[25]. A CFG is a directed graph where every edge represents a change in control flow within the computation and every node represents a basic block. This abstraction enables identifying loops through cycles, dead code analysis, and many other compiler optimizations by operating on the graph directly. Virtually every compiler today operates on a CFG. This includes MSVC, gcc, clang, and the Intel compiler.

LLVM, for example, is an IR framework that represents an application's code[26]. It is by default platform agnostic and represents programs as a traditional CFG. Ultimately, LLVM compiles down into assembly for various architectures through IR transformations. Currently LLVM supports primarily C and C++, but other languages have developed frontends such as Fortran, Julia, and Rust amongst others. This makes it a convenient platform for understanding how a computation is structured and a useful framework for operating on programs.

The concept of basic blocks are a required abstraction to discuss how programs execute[27]. A basic block represents a segment of code that must execute in order. Since the code is guaranteed to always execute in the same order, it is possible to abstract away the individual instructions to instead talk about the entire block as a unit. Modern CFGs utilize basic blocks as the nodes for exactly this reason. It is possible to represent everything done in a block by simply referring to it rather than each individual instruction in it. It is

important to note that basic blocks refer only to code that cannot branch internally. It is possible to split a basic block in half and each half would still be a valid block as long as a branch is inserted between them. Although this is the same computation as far as the processor is concerned, compilers often utilize this abstraction.

The final instruction in a basic block (the terminator) defines the control flow of a program. These are branches, jumps, function calls, or function returns. In general, function calls do not need to be at the end of a basic block, but this work artificially constrains it to make programs conform to a CFG, even across functions.

LLVM has a hierarchical structure that organizes code in a way most software developers can understand[28]. At the top is the Module, which simply represents a linked IR. Each module contains global variables and a list of functions. LLVM functions map directly to functions in C. C++ and other such languages also correspond, but their names are often mangled to provide additional language features. Each function is composed of a CFG that represents the control flow between basic blocks in the execution.

MLIR is an alternative program representation to LLVM that is intended to represent higher-level languages that are difficult to represent at the same level as LLVM[29]. Its exact grammar is variable and is allowed to vary as long as it possesses two properties: it must still possess functions, basic blocks, and instructions, and it must have a conversion grammar to allow it to convert to other MLIR grammars. These requirements force MLIR to maintain the same CFG abstraction that is used in most compilers. MLIR was in heavy development at the beginning of this work, but there is no fundamental change that would prevent it being used.

The techniques described in this dissertation rely upon a modified version of a CFG that makes it simpler to work with code that crosses function boundaries. Two modifications were made to the CFG: functions can only be called as the penultimate instruction, and

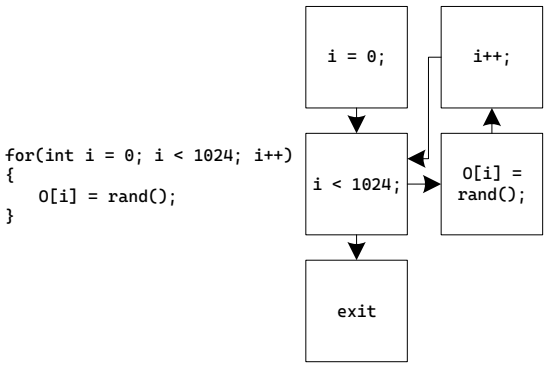


Figure 1: For-Loop CFG

C-style for-loops can be represented as four blocks within the CFG. The initial condition is the first block, in this case initializing i. The next block is the conditional. This block is responsible for defining the exit condition and is succeeded by both the body and the exit of the loop. It is simultaneously the first and last block that will execute as a part of the loop. In this case it checks if i is less than 1024. Next is the body block which performs the bulk of the logic in a for-loop. It can be a single block as seen here or lead to other computations including other loops before starting the incrementor. The incrementor is the final block in the loop and represents logic to be performed after the body, in this case just incrementing i. The incrementor is succeeded by the conditional where the loop begins again.

function calls and returns also form edges within this graph. This approach is similar to inlining, creating a single graph from the entire application. By making this transformation it is possible to describe the exact execution of code as a path within the CFG.

2.2 Loops are Kernels

Loops within code dominate the runtime of most applications[30]. Colloquially, we call these loops kernels. Most applications are littered with some type of kernel, but different domains prioritize particular subsets at the kernels they care about. Image processing kernels describe stencil based computations that stride across a buffer. Radio processing applications analyze temporal windows of data to translate it into other forms. Linear algebra kernels

specifically execute in matrix order. Although each domain prioritizes a different kind of kernel[31], they all refer to some type of computational loop.

There are a litany of divergent kernel definitions[8][32][33][34][35], but most of them devolve a kernel into two sections: the loop grammar and the core computation. The loop grammar is responsible for specifying how the kernel will loop through the computation. The core computation defines the actual arithmetic that a kernel performs. DSLs are built around a specific kernel definition, allowing them to make implicit assumptions about the memory grammar and the memory access pattern to simplify their representation. When the memory access pattern is ignored, most DSLs have similar structures. This is exploited by Delite[36], a DSL to define other DSLs. For more information about DSLs, see Section 2.3

The current approach to defining kernels relies upon enumeration of kernel archetypes. Asanovic et. al, for example, defined a kernel as code that conforms to one of twelve "dwarves"[7]. Since this definition is enumerative, these twelve categories do not represent all kernels. Instead, they were selected to represent the vast majority of computation. or example, one of the categories refers to spectral methods rather than a more specific, recognizable label such as FFT. Spectral methods is a superset which contains many subclasses, such as FFT. Other works have developed derivative kernel definitions[37], but to the author's knowledge there is no kernel definitions that does not rely upon enumeration.

A decidable kernel definition is required to make broader statements about the nature of kernels. This definition should be general enough to not exclude prior definitions. This definition should be decidable such that one can state affirmatively whether something is a kernel or not. This definition should conform to the community's intuition on what a kernel is. This work proposes such a definition in Chapters 3 and 4 that attempts to solve all these problems.

The memory grammar of a kernel reads from and writes to memory, producing a

dependency chain between kernels. This dependency chain will form a dependency graph between kernel instances. This dependency graph dictates the order in which a kernel must be executed to maintain the original functionality. This dependency graph is the primary aspect that limits a program's intrinsic parallelism.

Matrix multiply is a simple kernel that can demonstrate the fundamental kernel properties. The naive version is written as a triple nested for loop; thus, it is composed of three kernels. The innermost kernel is the simplest to discuss. The loop grammar is a simple iterator, $i = i + 1$. The memory grammar is also simple, in the case of row-major order $A = I_1[i][a]; B = I_2[a][i]$. The core computation is a simple accumulation, $O = O + A * B$. This is the type of terminology a DSL uses to represent kernels. Individual DSLs make assumptions about these properties, but they all operate on the assumption that there is a loop grammar, memory grammar, and core computation.

2.3 Kernel Transformations

Kernels are exploited today in two different methods: compilers utilize DSLs to better optimize kernel code, and hardware accelerators specialize in a particular kernel category to achieve higher efficiencies and speeds. Individual kernel categories possess specific memory and loop behaviors that enable computational shortcuts. If this information is known in advance, compilers and architectures can perform optimizations that are not possible on arbitrary code.

2.3.1 DSLs

DSLs are a programmatic representation of kernels that makes assumptions on the structure of the expressed kernels[31]. Traditional languages lack any method of specifying kernels, forcing them to fall back on loops with compiler directives[38]. Providing a language to represent a kernel is the first step in optimizing a kernel.

Delite is a framework that specializes in expressing DSLs[36][39]. Within Delite, ten distinct loop grammars which they call "Parallel Patterns." Delite differs from most other DSLs by allowing basic blocks to exist as a "sea of nodes" instead of in a traditional CFG. This was performed to better enable parallelism by having data be pulled from prior computations rather than pushed to subsequent ones. Delite still relies upon enumeration for its parallel patterns since it is compiling into a known form. It makes no claims on if other structures may be necessary.

DSLs have been applied most frequently to image processing kernels. Halide, for example, is a library that specializes in the expression of stencil computations[8]. It separates the design of the core computation from the loop grammar, while making the memory grammar implicit in the stencil operation. This makes parallelism easy to express for traditional architectures. In addition, by separating the arithmetic logic from the loop grammar, Halide allows for the separation of the algorithm design and loop scheduling[40]. This separation is a unique property to stencil computations that the Halide DSL efficiently exploits. Most DSLs rely upon a similar property to this to motivate their use.

By far, the most widespread DSL today is CUDA[41]. CUDA is a language that was developed by Nvidia to make it simpler to write vector code that can be executed on a GPU. Due to its widespread adoption, many do not view CUDA as a DSL. Despite this, CUDA is a language that was developed to help developers write code that targets a single domain,

GPUs. The development and adoption of this DSL has resulted in widespread utilization of GPUS for data parallel workloads. The development of a tool that makes it as easy to write accelerator code as it is to write CUDA has the potential to enable the adoption of a slew of new, efficient accelerators.

2.3.2 Hardware Accelerators

Various architectures have also been developed that exploit the unique memory access patterns present in a specific subclass of kernels. GPUs, the most prolific specialized architecture, were developed to exploit vector floating point calculations[42]. TPUs, a recent architecture developed for datacenters, exploit floating point matrix operations that are widespread in neural networks[43]. FFT accelerators have been developed to aid in radio processing tasks on edge platforms[44]. Linear algebra accelerators are commonplace to handle many different tasks more efficiently than a CPU can[45]. Each of these architectures is unique and has the potential to exploit a particular code structure more efficiently than any of its brethren.

Heterogenous systems were developed to provide a collection of accelerators that could be utilized by programs[46]. By identifying kernels in the wild code a layperson would write, it is possible to schedule arbitrary code on the accelerators present in a heterogenous system. This has the potential of both raising the efficiency of a computation while simultaneously completing it faster.

Heterogenous code execution is already in use today. Mobile devices already rely upon Big.little cores to delegate less time critical tasks to more efficient hardware[47]. CPU/GPU co-scheduling relies upon profiling the code to determine what is the most

efficient scheduling between architectures[48]. More advanced uses are in development now that allow for the use of several accelerators in the same system.

2.4 Synopsis

Kernels dominate our execution, but there are few ways to automatically optimize them to achieve top performance. Every computation of significant size is guaranteed to be dominated by some type of loop. Due to the design of C, the backbone of modern computation, there is no method to detect all loops or make inferences about the loops behavior. Some pattern matching techniques have been applied, but their application is limited and requires that a programmer write code with said framework in mind.

If kernels could be identified automatically, there are a plethora of ways this could push the field forward. The automatic detection of kernels could enable the solving the auto parallelism problem since kernels represent the primary memory path of the system. It could be possible to schedule code to run on a heterogenous accelerator by breaking a program down into its major constituent parts. It has the potential to steer the direction of hardware development by identifying which computations are the most important to accelerate from the code that is running. Identifying kernels by themselves provides very little, but it enables many transformations that could revolutionize computation.

Chapter 3

AUTOMATED COMPUTATIONAL KERNEL EXTRACTION FROM DYNAMIC APPLICATION TRACES

Compute kernels are the fundamental abstraction used to identify speedup opportunities, structure programming languages, and design hardware accelerators. Kernels are accelerated by increasing memory locality, increasing data-parallelism, and exploiting producer-consumer parallelism among kernels. Most data-parallel optimizations require that the kernels be presented in a recognizable form through annotation or a Domain Specific Language (DSL), but there is currently no method of extracting kernels from wild code. This paper describes a technique to automatically localize parallel kernels from a dynamic application trace, facilitating further code optimization.

Dynamic trace collection is fast and compact. With optimization, it only incurs a time-dilation of a factor of nine and file-size of one megabyte per second, addressing a significant criticism of this approach. Kernel extraction is accurate and performed in linear time within logarithmic memory, detecting a wide range of kernels. This approach was validated across 23 libraries, comprised of 31,263 kernel instances. To validate the accuracy of the detected kernels, the resultant kernels were analyzed for compatibility with our kernel layout. Over 97% of kernels detected from the test applications were fully compatible with the kernel definition proposed in this work.

3.1 Introduction

Continued performance scaling will require domain-specific architectures[49]. While it used to be possible to count on the continuous arrival of better switching devices, Moore's law has slowed, Dennard Scaling has ended, and a disruptive device is not yet commercially ready. While a sea of fixed-function accelerators appears attractive[50], the opportunity-cost and risk associated with specificity will force system-on-chip composition towards a modest-set of minimally-flexible accelerators[51] and traditional general-purpose processors.

A deep understanding of applications is required to design and utilize this cohort of processing elements (PEs) with varying degrees of flexibility focused in a subset of computation, called a domain-specific system-on-chip (DSSoC). Recurring motifs in the tasks composing applications in that subset will determine which PEs are selected and how they are tuned. Floor-planning, interconnect, and memory hierarchy will need to incorporate knowledge of those tasks' probabilistic relationships. Dynamic scheduling of tasks on a DSSoC platform is impossible if an application cannot even be factored into tasks.

As an abstraction for this application analysis, programs can be modeled as a producer-consumer graph of kernels. This work proposes a formal definition for kernels in Section 3.3 as a sub-graph of basic blocks that recur many times (eg. loops). This definition innovates over the example-driven definition provided by Asanovic [7] and the pragmatic definitions implied by the restrictions in data-parallel programming frameworks [36] by providing a decision function. As kernel basic-blocks contain load and store operations, they are data-dependent on the values produced by other kernels and have dependents that consume their work [52]. Thus, application understanding becomes a problem of segmenting basic-blocks into kernels and tracking their data interdependence.

Currently, kernel localization within an application requires manual program annota-

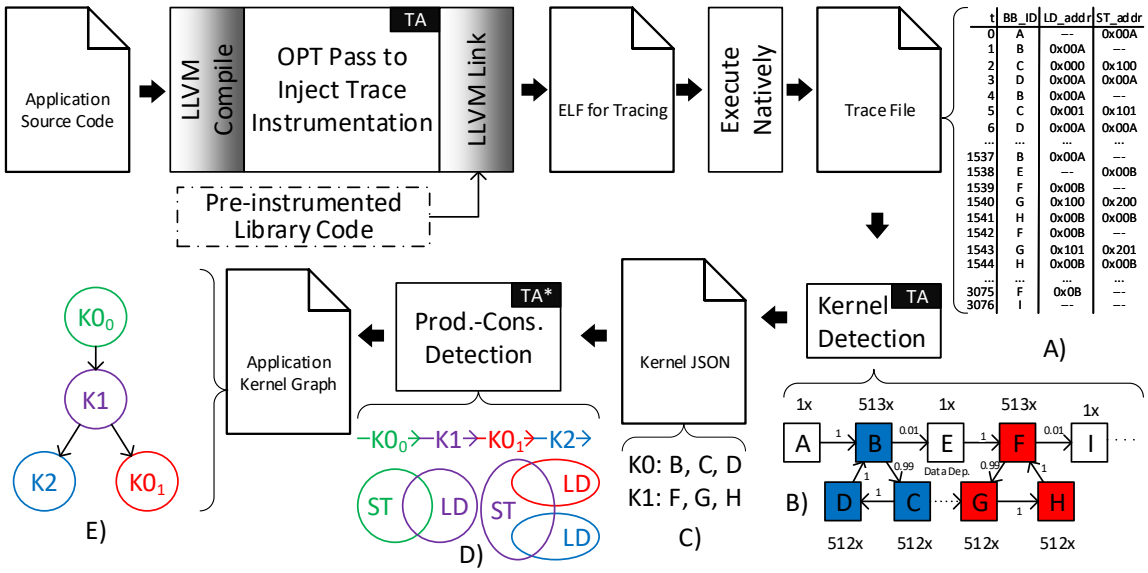


Figure 2: TraceAtlas Analysis Pipeline

Trace statements are injected into each basic block at the LLVM IR level. Each statement logs information about the basic block every time the block is executed, creating the trace (A). From the trace, kernels are detected based on temporal basic block adjacency and refined to produce executable kernels (B,C). Kernels are then analyzed in the trace to detect the producer-consumer relationships (D). The tools provided by TraceAtlas are annotated by a black "TA" box. Native LLVM tools are annotated with a gradient. The Producer-Consumer tool is not described in this work, but is available in the TraceAtlas repository.

tion. Static parallelization techniques, like those found in High-Level Synthesis (HLS) and polyhedral optimization, depend on a combination of compiler directives, code formatting, and programmer intention[38, 53]. Languages meant for programming data-parallel accelerators, like CUDA and OpenCL, imply a one-to-one mapping of functions to kernels. Domain-specific languages (DSLs) adopt this convention, limiting their expression to computations that are trivially introspected as data-parallel kernels with symbolic data-dependencies[8, 36]. While these solutions can provide incredible performance, the effort to transform or translate an application by factoring its kernels into these framework is limiting.

For a novel or naive application code, profiling drives this factoring process. A programmer could view clusters of hot-code as candidates for optimization[50]. If the data-dependencies between these clusters is indirect, a log of address values seen during execution will identify the overlap of kernel working sets. The combination of profiling, value logging, and recording sequence is dynamic-tracing. The increased visibility provided by tracing enables optimizations beyond those available to static techniques[54, 55]. Unfortunately, the excessive time dilation, disk-devouring trace-size, and poor analysis performance[56, 57] have limited the use of dynamic-trace techniques to the small-in-scale as in just-in-time compilation. Thus, to support code from the wild, any approach to factoring applications into graphs of kernels will need to address dynamic tracing performance.

This work addresses both concerns, factoring application code into kernels using a efficiently generated dynamic trace. As basic blocks represent a necessary ordered execution of instructions without branching, logging basic-block execution rather than instruction execution will result in a lossless order-of-magnitude reduction in trace effort. Further, many values are irrelevant to understanding the producer-consumer relationship (e.g., operands of arithmetic instructions), thus generating a parsimonious set of trace values (i.e., load and store addresses) creates a further work reduction. Basic blocks that recur with great frequency likely form a kernel with other blocks found nearby in time; however, as we describe later, clustered frequency on its own is not sufficient to construct a full kernel.

This paper describes TraceAtlas, an open-source[1] tool demonstrating the procedural extraction of kernels from unannotated applications expressed in LLVM-compatible languages. Figure 2 shows the TraceAtlas pipeline which injects function calls to produce a trace during execution, analyzes the trace to extract kernels, and combines the trace

[1] Available from CodeOcean at https://codeocean.com/capsule/2cb73b4e-11f9-4547-8fe3-4b4956d3d251/ and GitHub at https://github.com/ruhrie/TraceAtlas

with the kernels to generate a producer-consumer graph of kernels. As the field's conception kernel is varied and ambiguous while being critical for high-performance (Sec. 3.2), this work proposes a formal definition for kernels (Sec. 3.3). Intrinsics injected during compilation create the parsimonious dynamic application trace during native execution (Sec. 3.4). A log-space algorithm walks the execution trace to identify high-frequency basic-block clusters as kernels, correcting the kernel composition with a set of legalization steps (Sec. 3.5). TraceAtlas successfully identified over 31,000 kernels in 1,821 applications from 23 libraries, demonstrating a dynamic-trace speedup of $230\times$ and $49\times$ over naive and state-of-the-art implementations (Sec. 3.6).

This work's contributions are:

- A formal definition for parallel kernels
- A parsimonious dynamic application trace technique
- A kernel extraction and legalization algorithm
- TraceAtlas[1] an open-source implementation of this work

3.2 Background and Motivation

Kernels represent the vast majority of program computation and are one of the most important code segments to optimize. A kernel is composed of a collection of basic blocks in a program that are clustered temporally and recur many times. Currently, kernels are only detectable if the programmer wrote their code in a specific format that modern tools expect. Because of this, modern kernel-based tools require kernels to be written explicitly, either through code annotations or using DSLs. The objective of this work is to demonstrate a method of detecting and extracting kernel code from unformatted code. Analyzing a dynamic trace would allow for simple detection of these kernels, but current techniques

are inefficient to the point of failure on larger programs and often cannot trace an entire program. Current approaches to kernel optimization place restrictions on what code is a kernel.

3.2.1 Current Kernel Segmentation Techniques

One way to factor code is to segment it into high-frequency sub-programs. By optimizing the hot code, large performance improvements can be made. Greendroid[50] utilized a hot code detector to identify the portions of code which represented large portions of the energy consumption. By identifying and transferring this computation to specialized hardware, they were able to save significant energy.

Another approach segments the code into kernels and leverages libraries composed of example kernels. Libraries such as BLAS and FFTW[16] exist to provide simple, high performance versions of particular types. This also forms the foundation for Asanovic's kernel definition which defines a kernel as belonging to one of twelve dwarves[7]. Ultimately, this type of definition is restrictive, forbidding kernels which do not conform to our preconceived notions. To discover new kernels, a generative kernel definition is required.

To support a greater degree of expressive freedom, language like CUDA create a base abstraction where a function maps directly to a kernel; however, this abstraction is tied heavily to the GPU abstraction and may miss opportunities for inter-kernel optimization[41]. Similar paradigms such as OpenGL[58], OpenCL[59], and Gramps[52] fall victim to the same flaws.

Halide[8] and DeLite[36] present a kernel definition with domain specific abstractions. Halide kernels are defined as a collection of motifs that can be combined into formal templates. These templates can then be extended to any stencil based computation. DeLite

constructs a kernel from basic building blocks. Both of these techniques require the identification of the majority of motifs and building blocks. Ideally such components would be extracted from existing code or provided by domain experts.

Auto-parallelizing compiler passes could be viewed as a form of kernel factoring but often require "help" from the programmer to facilitate these transformations. Polyhedral analysis tools such as LLVM-poly[53] are able to detect loops in code through static code analysis. They take on the view that a kernel is a parallelizable loop in code. As a result, techniques such as these are limited to loop detection and will require that the loops be written in a way that the tool recognizes. HLS works around this by requiring that all loops of interest be manually annotated. This technique allows the programmer to specify exactly what should happen to their code, with the caveat that the user must know how to optimize the code themselves and what kernels are in their code.

3.2.2 Demand for Dynamic Kernel Analysis

Current static analysis tools fail to detect several important variants of kernels due to low level code representations having no limitations on memory accesses. Dynamic loop boundaries make it impossible to know how parallel a loop is without fully executing it. Recursion is another example where the number of recursions is not inherently obvious and may be difficult if not impossible to determine. An FFT for example will recurse down $\log_2 n$ times and this behavior cannot be determined without knowing n in advance, something that may or may not be known at compile time. Information critical to the detection of kernels is unavailable at compile time (i.e. static methods), and we must look to the runtime (i.e. dynamic methods).

Identifying kernels from hot code is insufficient due to the presence of low frequency

conditions in some kernel archetypes. To avoid this, current techniques require that each kernel be wrapped in a function, which is not how kernels exist in the wild. Additionally, not all hot code is a part of a kernel. To extract kernels from an input program, more information is required than simple execution counts.

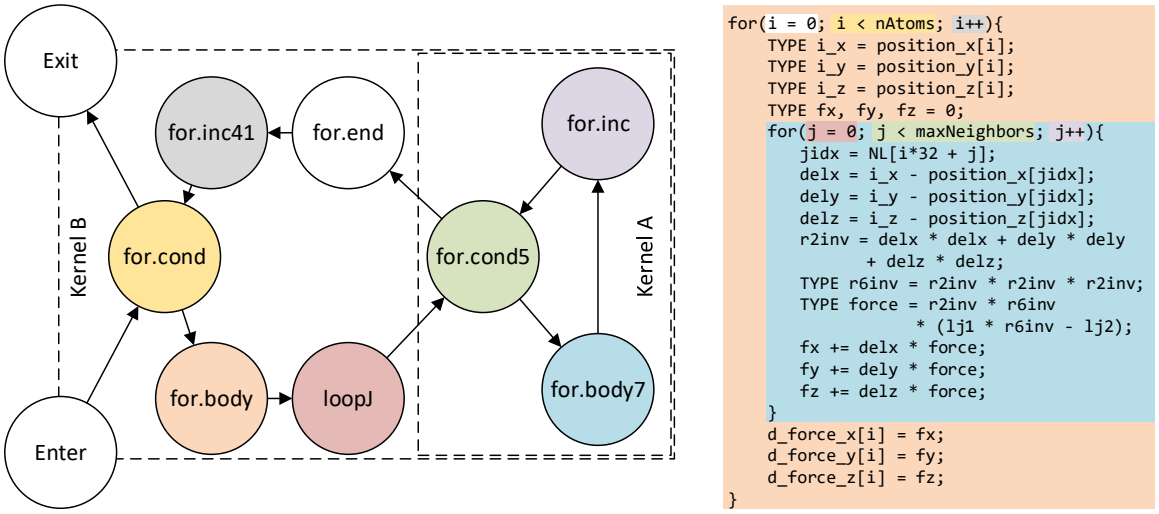
Aladdin, for example, extends the profile by using injected LLVM byte code to produce a history of the program at run-time[60]. TraceAtlas also performs this task, but the size of the trace grows quickly. Wasabi also generates a dynamic trace, but they are able to achieve a time dilation factor of 70 by tracing high-level WASM rather than the lower level operations[54].

TraceAtlas marries the two techniques to generate a dynamic trace with a low time dilation factor. This is accomplished by only storing the minimal information required to reconstitute the trace. This results in the generation of a parsimonious trace that can be used to fully reconstruct the execution of the original program.

A dynamic trace can provide the information that current static analysis tools lack. Currently, Aladdin limits their analysis to single kernel programs. This makes it trivial for them to detect the kernel and to make inferences about it. TraceAtlas addresses this limitation by providing a kernel definition and analyzing the trace to extract all kernel instances. Prior works have been limited in how they optimize parallel code by requiring the code to be formatted correctly and isolated. By extracting the kernels from wild code, these transformations can now be applied to general programs with parallel components.

3.3 A Generic Kernel Definition

A kernel is a collection of basic blocks that cluster temporally and recur many times. Basic blocks represent a series of instructions that must be executed in order. Subroutines




```
for(i = 0; i < nAtoms; i++){
    TYPE i_x = position_x[i];
    TYPE i_y = position_y[i];
    TYPE i_z = position_z[i];
    TYPE fx, fy, fz = 0;
    for(j = 0; j < maxNeighbors; j++){
       jidx = NL[i*32 + j];
       delx = i_x - position_x[jidx];
       dely = i_y - position_y[jidx];
       delz = i_z - position_z[jidx];
       r2inv = delx * delx + dely * dely
             + delz * delz;
       TYPE r6inv = r2inv * r2inv * r2inv;
       TYPE force = r2inv * r6inv
                  * (lj1 * r6inv - lj2);
       fx += delx * force;
       fy += dely * force;
       fz += delz * force;
    }
    d_force_x[i] = fx;
    d_force_y[i] = fy;
    d_force_z[i] = fz;
}
```

Figure 3: Kernel Basic Block Diagram with Code
The code on the right is a kernel that was detected from the MD benchmark in SHOC. The diagram on the left represents Decision-to-decision path (DD-path). Each individual basic block is highlighted in a matching color to indicate where the basic block derives its code from. Code that is white or a basic block that is white represent a feature that has no analog in the other domain.

(i.e. functions and inner-loops) are composed of multiple basic blocks executed in sequence, and a kernel is represented by their collection. Given the sequential execution of these subroutines, basic blocks that are executed close-in-time are probable to be members of the same kernel.

A program is represented as a directed graph of basic blocks, where the directed edge represents succession in execution. This abstraction is utilized by the entire TraceAtlas toolchain and is a modified version of a Control Flow Graph (CFG). It varies from a traditional CFG by incorporating function calls as regular edges. To make this legal, function calls are limited to be the penultimate or ultimate instruction in a basic block. With this limitation, an edge can be drawn from the call site to the entry block and from the return blocks to the function call's successor. If runtime information is available, a weight can be

attached to each edge that refers to the total number of times that path was taken. For the duration of this paper, this abstraction will be refereed to as G_c.

A kernel can be defined within this abstraction through three constraints:

1. Each kernel is a strongly connected subgraph
2. Each kernel may not partially overlap, $G_A \cap G_B = \emptyset$ or $G_A \subset G_B$
3. Every block within a kernel must be more likely to select a successor block within the kernel set than a block outside of it

3.3.1 Constraint 1: A Kernel is a Strongly Connected Subgraph

All blocks must be composed of some repeating computation. Within the graph, a recurrence requires that any block be capable of repeating the computation. This implies that there must be a path from any block in a kernel to any other block in the kernel, also known as a strongly connected subgraph. An example of the kernel structure can be seen in Figure 4.

This work defines a strongly connected subgraph to be a subgraph where $\forall(U, V)$ there is a path from U to V where U and V are vertices in the TraceAtlas abstraction. The length of this path is also restricted to not be zero. Do note that this work allows U and V to be coincident, permitting single node subgraphs as long as there is an edge from a node to itself. This definition makes it such that each basic block must have the ability to reenter the loop and traverse a different path through the computation. Every recurrence in code must create a cycle. Since every cycle is implicitly a strongly connected subgraph, every recurrence must also form a strongly connected subgraph. Figure 3 is an example of this. Both Kernel A and B are composed of nodes that form a strongly connected subgraph.

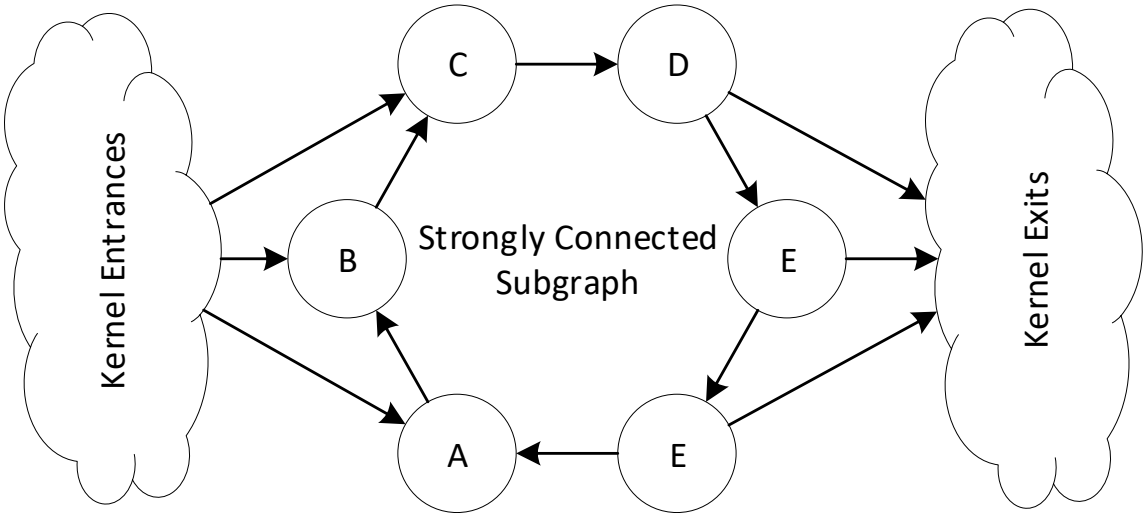


Figure 4: TraceAtlas Kernel Graph

TraceAtlas creates kernels that are composed of at least one cycle in the computation. The "cycle" that defines the kernel is a strongly connected subgraph in the computational graph. Each edge that enters the strongly connected subgraph is a kernel entrance. Each edge that exits the strongly connected subgraph is a kernel exit. There may be several kernel exits and entrances. Additional nested kernels may exist within this graph as a strongly connected subgraph within the parent kernel.

3.3.2 Constraint 2: Kernels May Not Partially Overlap

Not every strongly connected subgraph is a kernel. There are potentially an exponential number of different strongly connected subgraphs in an arbitrary graph. To limit how many of these subgraphs are legal kernels, an additional constraint is required. To conform to the colloquial definition of a kernel and the spirit of the first constraint, a kernel must contain the entire loop. If a kernel overlaps with another, it can be inferred that not every path is contained within the kernel. Thus, a kernel must either fully overlap with another (in the case of hierarchy) or not overlap at all. This is the cause of Constraint 2.

This can be expressed though a simple set relation. Take graphs G_{k1} and G_{k2} as strongly

connected subgraphs of G_c. To satisfy the constraint, $G_{k1} \cap G_{k2} = \emptyset$ or $G_{k1} \subset G_{k2}$. This makes it trivial to identify malformed kernels.

Figure 3 contains a strongly connected subgraph that fails this criteria. By excluding $for.body7$ and $for.inc$ a strongly connected subgraph can be formed that overlaps Kernel A. Determining which kernel is correct, if either, requires runtime information to determine which edges are traversed and how frequently they are.

3.3.3 Constraint 3: A Kernel Must be More Likely to Recur

A kernel must be more likely to continue executing than to leave at any point. This is the distinguishing characteristic between a kernel and an arbitrary strongly connected subgraph. The simplest way to quantify this would be to attach the execution count of each edge as a weight in the graph as is done in G_c. The probability of remaining within the subgraph would then be represented by the ratio of the sum of edges leaving the kernel and the sum of edges remaining in the kernel. To be more probable, this number should be greater than one-half.

Utilizing this probability information, the constraint can be checked through the properties of the graph. To do so, define G_k as the kernel subgraph and $G_{\overline{k}} = G_c - G_k$. Define E_c as the edge cut[2] of G_k and $G_{\overline{k}}$ and E_k as the edges in G_k. Define N as the nodes in G_c. This constraint states that $\forall N \in G_k$, the sum of all edges in E_c that are incident with N must be less than the sum of all edges in E_k that are incident with N. Being a digraph, this can be performed on incoming edges, outgoing edges, or both. Ultimately, they form the same constraint so this work will apply this constraint to both edges. This guarantees that a kernel is more likely to select a successor in the kernel than from outside it.

[2]the edge cut is defined as the list of all edges connecting the two subgraphs

3.3.4 Implications of These Constraints

Should one kernel be a subset of another, then the kernels form a hierarchy. This happens most frequently with nested for-loops (see Figure 3). Thus, kernels can nest within each other. This hierarchical representation only refers to how kernels are expressed in code. Just like there are grammars that permit reordering loops, a grammar could be developed to manipulate hierarchical kernel relationships while maintaining identical functionality.

Representing code as it is found in the wild requires permitting multiple entrances and exits from kernels. Vertices with an edge originating outside the kernel subgraph are *Entrances*. Vertices with an edge terminating outside the kernel subgraph are *Exits*. Because kernels must be a part of the program, there is at least one Entrance and one Exit. The vast majority of kernels only possess one of each. Multiple exits occur occasionally, with the most common source being break statements. Multiple entrances occur when a loop can be entered later on. These are rare and not possible in C without jumps, but compiler optimizations can create such structures in LLVM IR. Most DSLs make such expressions illegal, but to represent code as it is currently written, it is necessary to include them.

The structure of the graph creates a pattern of execution. Implicit in this pattern, basic blocks in these graphs are likely to occur temporally adjacent to each other. This is demonstrated in Figure 5 where each edge represents the probability of a basic block following that path. By taking advantage of the basic block adjacency, a greedy algorithm can be developed to detect kernels from dynamic application traces. With the help of some refinement algorithms, the kernel structure can be easily extracted from naive, wild code. This will be discussed in Section 3.5.

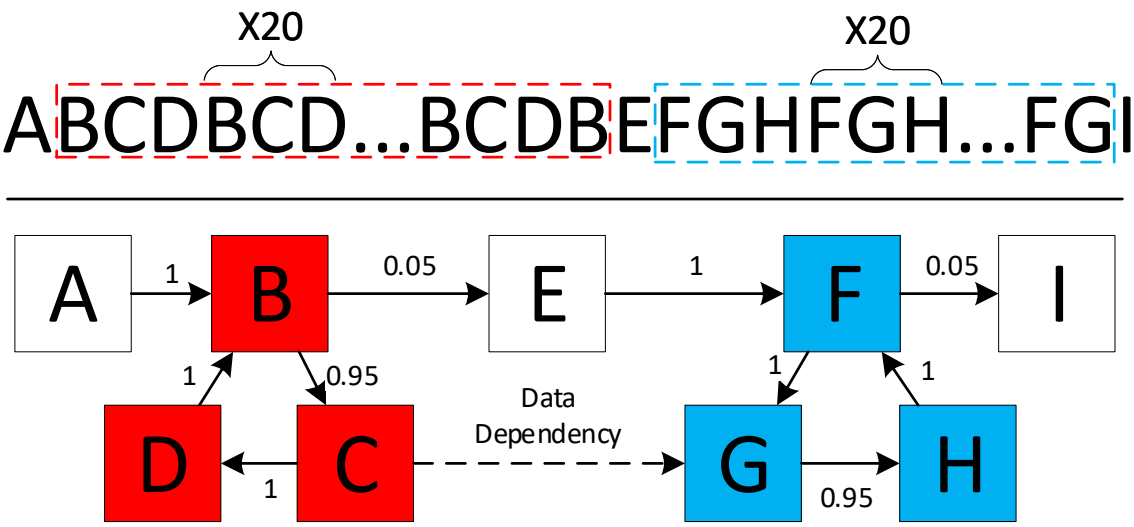


Figure 5: Kernel Basic Block Example
Above is an example temporal ordering of basic blocks in a dynamic trace. Below is the decision-to-decision path (DD-Path) for the computation where each edge weight is the probability of the computation following that path. The first kernel in red is composed of a for loop and repeats twenty times. The second kernel in blue is a recursion that also occurs twenty times and consumes data from the first kernel.

3.4 Low Overhead Dynamic Traces

Dynamic tracing is prohibitively expensive in both runtime dilation and disk utilization without a pre-analysis step that may discard relevant information. Generating an entire trace is required to avoid preconceptions on the structure of the input programs. A naive trace which only stores executed instructions produces a large amount of data, often exceeding 100 gigabytes for a single-second program. A few simple, high-level optimizations dramatically reduce the required amount of space. This paper evaluates the following optimizations:

- Compressing the trace data with Zlib[60]
- Clustering the trace writes to lower OS overhead
- Encoding trace information before it is compressed with Zlib

- Adding memory tracing with these enhancements

These optimizations shrink the execution time to a runtime factor of nine and produce a trace that rarely exceeds five gigabytes. Table 2 demonstrates the relative improvements and cost of each optimization. Simple compression, as is done in Aladdin[60], makes a much smaller trace, at the expense of it taking approximately twice as long. Bursting the data export saves 21.4% of the runtime at the expense of a 6% space overhead. Encoding the information before it is traced raises the density of the information dramatically. With this optimization, the runtime overhead is merely a factor of 9. Adding address information to the trace significantly raises the tracing time and size (by a factor of 27 and 7.8 respectively), but this could likely be improved through the use of memory stride analysis[61]. Since this work did not use the traced memory, this optimization was not explored.

TraceAtlas generates dynamic traces by injecting logging statements into the LLVM IR. The naive solution is to simply export the IR as the application executes. This generates a trace in plain text and is relatively robust; however, applications today run billions of operations a second. By exporting all information with no processing, the disk always becomes the bottleneck in both write speed and storage size.

Trace data can be compressed as it is generated by Zlib. This is the current state of the art solution used by Aladdin for potentially unstructured data[60]. Trace information is low in information and can thus achieve exceptionally high rates of data compression. This

Table 2: Normalized TraceAtlas Performance Comparison

Technique	Size	Time	Effective Change
Naive	40.8	238.8	Raw dumping
Zlib (SoA)	1.704	438.1	Compress output
Bursting	1.808	348.98	Burst dumps
TraceAtlas	1	9.1	Encode information
+Addresses	7.84	248.9	Add addresses

optimization moves the bottleneck from disk write speed to processor execution. Due to the high repetitiveness of the data, Zlib was able to easily compress the traces. Different compression levels did not appear to appreciably impact trace size or time. Overall, this optimization was found to double the execution time, but shrink the trace size by 20x on small applications. Larger applications dramatically increase the runtime due to Zlib operating better on smaller kernels. Numbers are unavailable due to the inability of naive tracing to produce a trace that fits on the hard drive, but intermediate applications had an overhead ranging from 500-2000x relative to the original execution.

Exporting information to the trace as soon as it becomes available is relatively expensive due to the system call overhead. There are two primary overheads in dumping the trace information: the call overhead to export the information itself and compressing the data with Zlib to write it to disk. Since basic blocks always executed in order, it is possible to only export the trace data periodically. This removes many potential export calls that scale with the size of the basic block. Similarly, Zlib compresses most efficiently when it has a relatively large chunk of data to process. The combined effect of these two optimizations resulted in a 25% trace time improvement.

Ultimately, compression relies upon encoding the information in a more succinct form. A more efficient identifier for the basic blocks is know at compile time, the basic block ID. If only the block ID is exported, significant performance improvements can be achieved. By exporting the block ID instead of each individual instruction the actual IR, the compression effort is significantly reduced. This efficiently encodes the trace before it is compressed while maintaining critical information on the structure of the computation. This results in the trace size halving again while the total time to trace shrinks to an overall time dilation of nine.

Additional information may be required beyond execution order that adds additional

overhead. Detecting kernel relationships, for example, requires looking at the memory dependencies, see section 3.7. To do this, the addresses of all loads and stores can also be exported. This results in the trace time rising dramatically and a significantly larger trace file. Aladdin's implementation exports the addresses of all load and store operations, but an in-house variant was written with support for additional features. As a result, addresses are not represented in the Zlib row reported in Table 2 which would more closely align with Aladdin[60].

TraceAtlas has minimal overheads and can trace every application with LLVM IR. Table 2 contains an overview of TraceAtlas's performance improvements with the average trace size and average time dilation factor for all applications that were able to be traced naively with less than a terabyte of disk space and with compression enabled in under 48 hours. The cumulative effect of all optimizations resulted in a runtime speedup of 400x on small programs versus SoA and over 100,000x for larger programs. Trace sizes halved for smaller programs and 1500x for large programs. These levels of overhead put execution time of an application to nearly real time. As a result, traces can be generated quickly with minimal data stored. This alleviates the costs of dynamic tracing and makes it a viable tool to perform additional optimizations, such as the identification of kernels.

3.5 Kernel Extraction

Detecting kernels from a dynamic trace is a multistep process. The initial kernels are detected from the trace based on their temporal affinity (Greedy Kernel, Step I). This is inspired by Greendroid and identifies potential kernels through hot code. The trace is then iterated through to add low frequency blocks (Steps II and III). The computational graph each kernel forms is then trimmed by removing non-recurring blocks removed from the

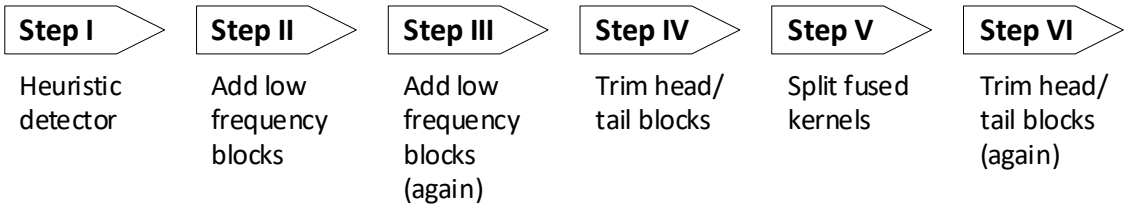


Figure 6: Kernel Detection Analysis Flow
Kernel are extracted by first using a heuristic detector on a dynamic trace. This identifies potential kernels through hot code and corresponds to Constraint 3. The the execution path is then followed for each kernel to fuse in low frequency blocks. This step happens twice and corresponds to Constraint 2. Afterwards, the kernels are compared to the source code to remove non-recurring blocks from the front and back of the kernel. The kernels are then analyzed to split fused recurring sections. Finally, the non-recurring blocks are trimmed again to maintain kernel structure. These final three steps all correspond to Constraint 1.

head and tail (Trimmed Kernel, Steps IV and VI). Finally, erroneously fused kernels are split by the elimination of non-recurring body blocks (Split Kernels, Step V). The Traced, Trimmed, and Split kernel extraction steps all refine a previous set of kernels and are chained together. By executing each of these steps as shown in Figure 6, kernels which conform to the structure in Figure 4 are produced.

3.5.1 Initial Kernel Detection

Greedy Kernels are extracted by clustering the basic blocks from the source application into a series of temporally adjacent basic blocks using a greedy algorithm. If a single basic block from a kernel within a trace is examined, there is a high probability that the preceding basic blocks will be a part of the same kernel due to how kernels "loop". Similarly, basic blocks that succeed a given basic block will also be probable to be a part of the same kernel

This work encodes the probability of temporal adjacency as an affinity, $P(A\alpha^r B^k)$ where α is an operator. This equation translates to the probability that basic block B occurs

k times within a distance r of A. Distance refers to the number of basic blocks that are executed between the current basic block and the target basic block. More generally, basic block affinity is the probability that any one basic block occurs within a range of another.

Using this affinity metric, a score can be calculated between any two basic blocks by summing over the size of the window as is done in Equation 3.1. The final score is subsequently divided by the size of the window to make the final scores mimic a probability and approach 1. For example, take the basic block sequence $A \rightarrow B \rightarrow A \rightarrow B \rightarrow A$. With a distance of 2, the affinity between A and B is 0.4 based upon the second A. Similarly, the affinity between A and A is 0.6. A kernel will then be defined as a union of basic blocks such that the mutual affinity of all basic blocks are greater than a specified threshold.

$$f_r(A, B) = \frac{1}{2r+1} \sum_{k=1}^{2r+1} P(A\alpha^r B^k) \tag{3.1}$$

This approach is advantageous because it only requires storing the previous $2r + 1$ basic block IDs in memory. Due to the potential of traces to contain billions of basic blocks or more, a logarithmic or constant space algorithm is mandatory to analyze the trace. This approach has linear temporal complexity with respect to the size of the trace. Its spatial complexity is technically logarithmic due to it simply counting the number of times a basic block occurs, but each count is implemented as a 64-bit integer. This makes the algorithm effectively consume linear space with respect to the number of unique basic blocks. Since the trace size is dramatically larger than the number of basic blocks in an application, the space consumed is constant, and Equation 3.1 satisfies this constraint. Figure 8 demonstrates the pseudocode to calculate this value.

With the affinities calculated, one can then represent the collection of basic blocks within a program as a fully connected, directed graph with edge weights equal to the given affinity. Figure 7 contains an example set of values where each edge is the maximum of the

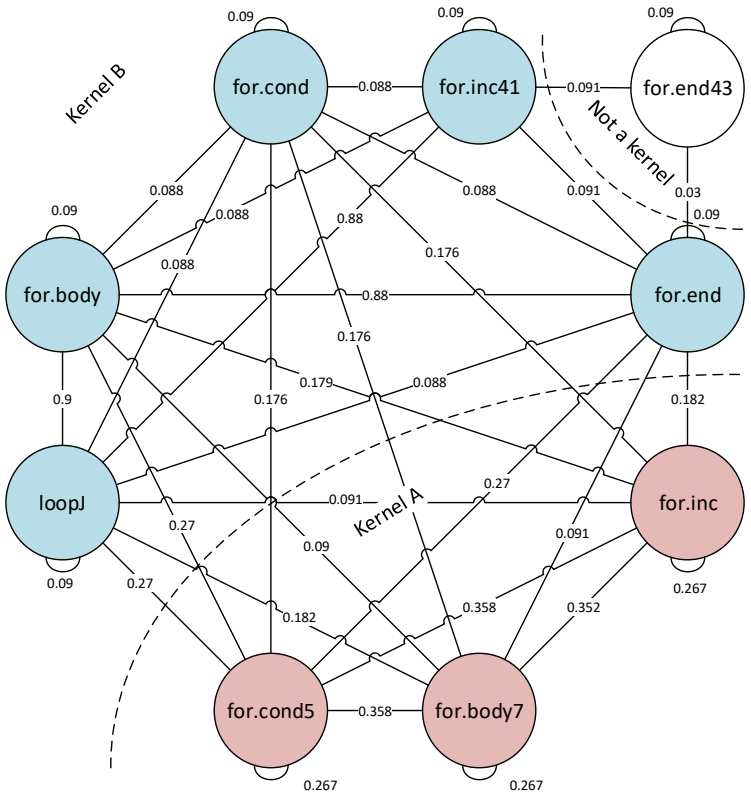


Figure 7: Kernel Probability Graph
The clique affinity graph is built from basic block affinities. Each edge represents the affinity between the basic blocks. This code originated from the MD benchmark in SHOC.

two edges between the nodes. Within Figure 7, each kernel is composed of an undirected graph where, from any given node, all edge weights sum to the target threshold, 0.95 in this scenario.

The graph is then segmented by utilizing a greedy algorithm as given in Figure 9. To do this, the algorithm iterates through basic blocks, using them as a seed to create kernels. The algorithm selects the block with the highest execution count that hasn't already been added to a kernel and uses it as a seed for a new kernel. Then, the algorithm greedily adds the basic block with the highest affinity score to the new kernel until a desired threshold is

```
for block in trace:
  for i in 1 to (2r + 1):
    bCount[oldBlocks[r]][oldBlocks[i]] += 1
    oldBlocks.shiftBack(block)
  counter[block] += 1
for block in bCount:
  for other in bCount:
    prob[block][other] = \
      bCount[block][other] / counter[block]
```

Figure 8: Basic Block Affinity Calculation
This code efficiently reads in a trace and calculates the affinity calculation from Equation 3.1 for every basic block in the trace. It has been tested on traces over a terabyte in size and executes in under a day.

achieved, currently 0.95. The resultant collection of basic blocks will be such that there is at least a 95% chance that a successor block to a block in the kernel will also be in the kernel.

3.5.2 Kernel Refinement Algorithms

The kernels identified by the greedy algorithm are merely an abstract set of basic blocks that have a high probability of being temporally adjacent. They lack any form of structural verification and are not structured like real code. To compensate for this, a series of refinement algorithms were developed. Traced Kernels add in missing basic blocks from the trace. Trimmed Kernels remove non-recurring blocks from the head and the tail of the kernel. Split Kernels remove non-recurring blocks from the body of the kernel. Each of these steps are necessary to refine the detected kernels into the form given in Figure 4.

Traced Kernels are created by identifying low-frequency blocks that were executed in a kernel in the trace and adding them to the kernel. To do this, the algorithm iterates through the trace, storing the blocks that have been executed since the kernel last executed. If the

```
explainedBlocks = set()
blockCount.sort()
for block, count in blockCount:
  if count < 512:
    break
  if block in explainedBlocks:
    continue
  sortedRow = prob[block].sort()
  kernel = []
  score = 0
  while score < 0.95:
    kernel.add(nextBlock)
    score = GetScore(kernel)
  for a in kernel:
    explainedBlocks.add(a)
```

Figure 9: Basic Block Fusion

With basic blocks affinities calculated, each basic block above the hot code threshold needs to be clustered to form kernels, 512 in this example. This listing greedily adds basic blocks to a kernel until the mutual affinity of all basic blocks in the kernel is greater than the given threshold, 0.95 in this example.

trace re-enters the kernel before entering another kernel, these blocks are a low-frequency part of the kernel and should be added. If a new kernel is entered, the current kernel must have been exited and blocks should be forgotten. Pseudocode of this algorithm is available in Figure 10.

This algorithm utilizes the entrance of the kernel to reset the blocks. It is possible that the kernel detected by the Greedy Kernel detector lacks the basic block for the true entrance. To compensate, this algorithm should be run twice where the second run has the correct entrance available. The first run will add in most of the low frequency blocks. If the entrance is missing, it is added during the first pass, making the second pass able to collect the remaining missing blocks.

```
newKernel = kernel
newBBs = set()
if bb == start:
    newBBs.clear()
else:
    newBBs.add(block)
    if block in kernel:
        if start == None:
            start = block
        else:
            newKernel += newBBs
            newBBs.clear()
```

Figure 10: Kernel Smoothing Code
This listing smooths a kernel such that the basic blocks contained represent a proper calculation. By tracking when you enter a kernel, new basic blocks can be added that match the path of the execution until it eventually leaves the kernel. This fully recreates the path the kernel took in the trace.

Trimmed Kernels are extracted by removing non-recurring blocks from the head and the tail of the kernel. These blocks are often added by the Greedy Kernel detector and must be trimmed such that they fit the model in Figure 4.

The kernels created by the Greedy Kernel detector have the potential to add in basic blocks on the front and the back of the computation such as in Figure 11. These basic blocks have an affinity that is high enough that they are selected for merging by the greedy algorithm, even though they are not a part of a strongly connected subgraph. This is particularly problematic with higher thresholds, but it was found that lower thresholds caused more systemic issues in the detected kernels, such as missing a block in the body. Head and tail blocks can be easily removed though, so higher thresholds are preferred.

Non-recurring blocks on the boundaries of kernels are removed by examining every node in the computational graph. By examining the sources and destinations of edges in

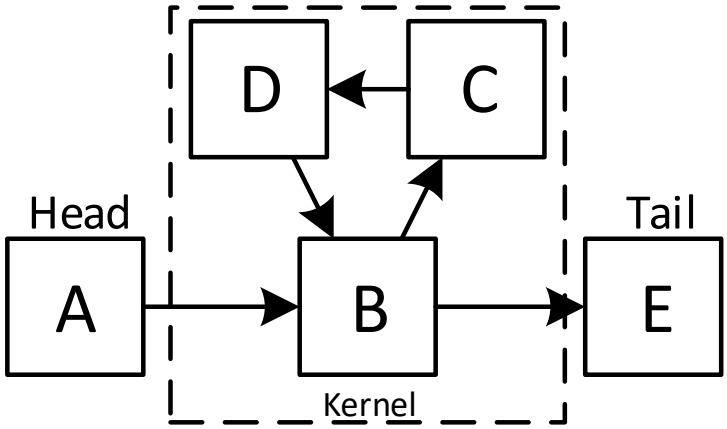


Figure 11: Trimmed Kernel Structural Flaws
Trimmed Kernels fix the kernel graph by removing non-recurring blocks from the beginning and the end of a kernel that are erroneously added by prior steps. Block A and E in this example cannot reach themselves and are therefor not a part of the kernel.

```
change = True
while change:
    change = False
    for block in kernel:
        for predecessor in block:
            if not predecessor in block:
                kernel.remove(predecessor)
                change = True
        for successor in block:
            if not successor in block:
                kernel.remove(successor)
                change = True
```

Figure 12: Trimmed Kernel Pseudocode
The Trimmed Kernel detector works by iteratively removing blocks that cannot reach themselves, and if there was a change it repeats until no blocks were removed

the graph, invalid blocks can be removed if the basic block doesn't have a successor or a predecessor in the kernel. The pseudocode for this process is available in Figure 12.

Split Kernels are detected by removing every basic block that is not part of a strongly

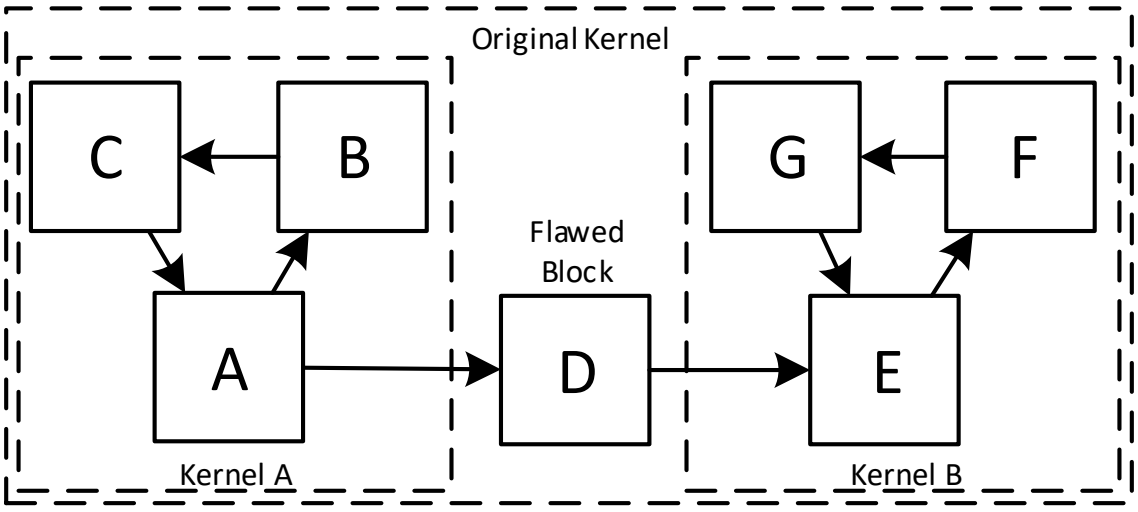


Figure 13: Split Kernel Flaws
Split Kernels fix two kernels that have been fused erroneously by an intermediate block. The original kernel here was the union of Kernel A, Kernel B, and block D. Block D cannot reach itself, and must be removed before resegmenting the graph.

connected subgraph. The remaining blocks are then used as seeds. Every block will be used to create a kernel by identifying every block it can reach. Because each kernel is a strongly connected subgraph, each block will create identical kernels, allowing for easy duplicate removal. This segments any kernel that has been fused erroneously.

Figure 13 demonstrates a graph that represents this issue. Two or more kernels are fused by the Greedy Kernel detector greedily adding blocks to reach the threshold. The Split Kernel identifies any basic block that cannot recur by exploring the graph and identifying every block a particular block can reach. If it can't reach itself, this is a fusion of kernels that must be split by removing the non-recurring blocks and segmenting the graph. The pseudocode for this is available in Figure 14.

This process is the only kernel step which can create new kernels. Every prior step is

```
def GetReachableBlocks(block, kernel):
  reachable = set()
  toProcess = set()
  toProcess.insert(block)
  while not toProcess.empty():
    for successor in toProcess.front():
      if not successor in reachable \
        and successor in kernel:
        reachable.insert(successor)
        toProcess.insert(successor)
    toProcess.pop()
  return reachable

def SplitKernel(kernel):
  for block in kernel:
    reachable = \
      GetReachableBlocks(block, kernel)
      if not block in reachable:
        kernel.remove(block)

  newKernels = set()
  for block in kernel:
    reachable = \
      GetReachableBlocks(block, kernel)
    newKernels.insert(reachable)
  return newKernels
```

Figure 14: Split Kernel Pseudocode

Split Kernels are created by identifying which blocks every block can reach. If it can't reach itself, it is removed from the kernel. The graph is then segmented by getting all the blocks a block can reach and creating a unique kernel for each of them.

reductive and will slowly remove erroneously detected kernels. This detector will remove incorrect kernels and replace it with multiple corrected kernels.

The kernel extraction algorithm takes approximately the same amount of time as it takes to trace the original program. It is simply not viable to have the program analyze the entire trace at once due to the oppressive memory overhead that would entail. To compensate, the algorithm operates on smaller segments of the trace. This requires that it iterate over the trace three times, once for the Greedy Kernels and twice for the Traced Kernels.

3.6 Results

TraceAtlas is an open source tool that can successfully dynamically trace an application with the primary limitation being the runtime of the program. TraceAtlas has an average runtime dilation factor of nine and produce one megabyte for every second of execution. The resultant traces can be efficiently analyzed to detect kernels using a greedy algorithm to identify the flavor of kernels of interest to the user.

3.6.1 Methodology

All of the applications traced for this paper were compiled with clang 9.0 and linked with lld 9.0. Zlib 1.2.11 was used for compression. Table 3 enumerates the collection of libraries and benchmarks used for tracing, the applications using that library, the number of kernels detected, and the version of the library or benchmark. The libraries were selected to sample a variety of general kernel-based tools. Over 31,000 kernels were identified from 1,821 programs originating from 13 open source projects.

The types of applications were selected to demonstrate the ability of the algorithm to

Table 3: TraceAtlas Test Corpus

	Library	Apps	Kernels	Version
Image	Halide[8]	41	977	2019/08/27
	OpenCV[18]	74	6,827	4.1.0
	FFmpeg[62]	9	66	4.2
Benchmark	FEC[13]	8	167	3.0.1
	MiBench[14]	27	170	1.0
	Perfect[12]	51	264	1.0.0
	Dhrystone[17]	7	74	2
	SHOC[15]	10	35	1.1.4
Math	VOLK	57	129	2.0.0
	FFTW[16]	84	4,686	3.3.8
	Armadillo[24]	184	1,496	9.600.6
	Cortex Suite[20]	19	702	2019/10/30
	Eigen[21]	47	2,958	3.3.6
	Gnu Scientific[11]	264	3,157	2.6
DSP	spuce[22]	93	849	0.4.3
	SigPack	56	2,243	1.2.4
	LiquidSDR[23]	114	1,409	1.3.1
	mbedTLS[19]	70	1,902	2.16.3
	StreamIt[10]	16	137	1.6
	Custom Naive Kernels	590	3,015	
	Total	1,821	31,263	

detect kernels. The known location of the kernels aided in the development of the algorithms to certify their accuracy. Although only 3,000 of the kernels in the test set were of entirely custom code, TraceAtlas still successfully identified and extracted kernels from them.

The applications were run on Xeon E5-2650 processors with the trace data being written to an Intel SSD DC S3500 with 1TB of storage. Each application was executed nine times and the median value was reported to filter out noise. Each sample was given 48 hours to execute and was allowed to consume an entire terabyte of storage. The application samples that exceeded these limits were terminated and are not present in Table 2 or Figures 15 and 16.

3.6.2 Dynamic Tracing

The optimization techniques resulted in a reduction of trace size by a factor of fifty relative to naive techniques and a reduction of trace time by a factor of two relative to Zlib compression. Each strategy approached a different problem and achieved a speedup in its domain and often for others as well. The net improvement of all these optimizations has resulted in a tracing technique that runs within a reasonable time dilation.

The TraceAtlas tool has the ability to enable the tracing of memory accesses. Of the techniques labeled in Table 2, only one encodes address information and has a higher cost associated with it. The current version of TraceAtlas has no currently known limitations beyond only supporting single threaded applications and the performance overhead of tracing.

Figure 15 shows that TraceAtlas, as a rule, produces smaller traces due to the information being compressed statically before being fed to Zlib. Occasionally, raw Zlib compression and IO clustering will produce a smaller trace, likely due to the method Zlib uses to compress the trace being more efficient that the current encoding scheme. For additional details, see Section 3.7. Table 2 shows that on average, TraceAtlas produces a trace that is half the size of what is produced from raw Zlib compression.

Figure 16 shows that TraceAtlas performs above average across the space; however, once it takes more than a second to trace, TraceAtlas performs significantly faster. When the trace takes TraceAtlas more than a second, it had a time dilation factor of 10.18 while Zlib compression had a factor of 265.5. This shows that TraceAtlas is twenty-six times faster than Zlib compression for larger applications. For additional analysis see Section 3.7.

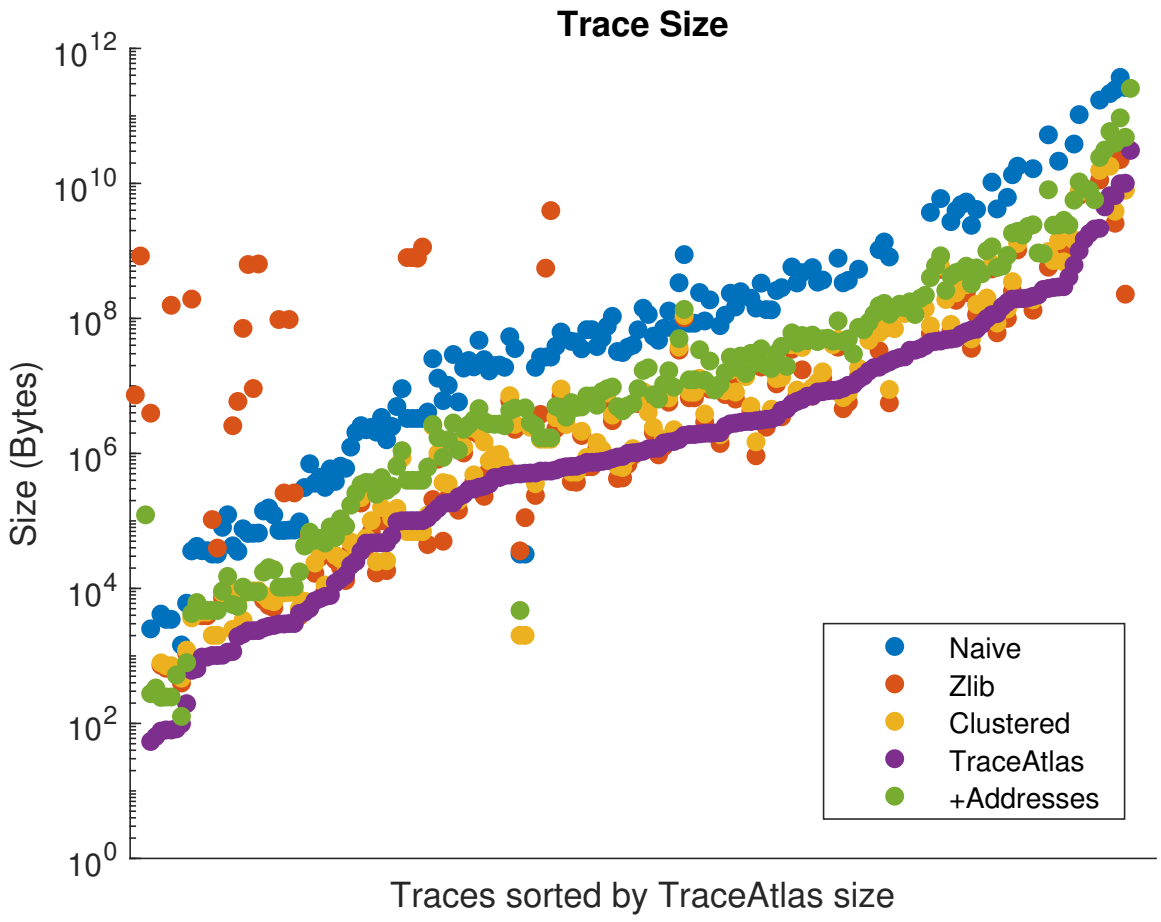


Figure 15: Overall Trace Size
The horizontal axis sorts traces by the size of the TraceAtlas trace. Usually, TraceAtlas is significantly smaller than other techniques. On occasion other techniques will perform better due to the implementation of Zlib, but on average it produces traces that are significantly smaller.

3.6.3 Kernel Extraction Algorithm

The kernel extraction process has two parameters: the affinity threshold and the hot code threshold. Figure 17 shows the compliance rate of kernels extracted with these parameters. Compliance is defined as a kernel which conforms to the definition given in Section 3.3. In short, every block must be able to reach itself, can only be reached through an entrance in the body, and conforms to the given graph.

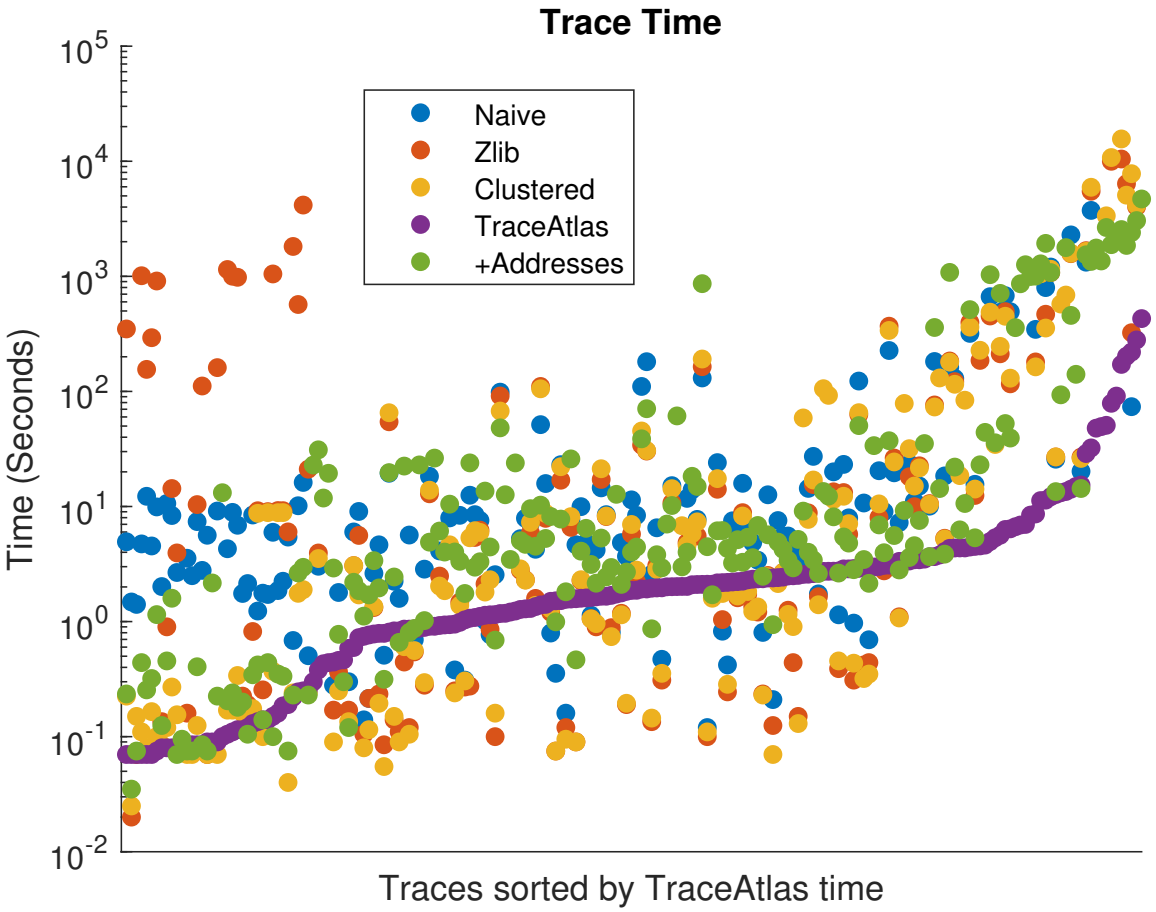


Figure 16: Overall Trace Speed
The horizontal axis sorts traces by the speed of the TraceAtlas trace generation. Usually, TraceAtlas is significantly faster than other techniques. On occasion other techniques will perform better due to the implementation of zlib, but on average it runs faster. Larger applications, however, perform dramatically worse than TraceAtlas.

Higher affinities raise the success rate, but due to the extraction process, compliance is greater than 98% in all cases. Similarly, higher hot code thresholds raise the success rate due to it creating a closer temporal affinity between blocks. The choice of these parameters do not significantly impact the validity of the algorithm. Instead, they should be selected to represent the types of kernels the user wishes to discover. Further refinements to the extraction process may make the specification of these parameters unnecessary.

Figure 18 demonstrates the median number of kernels detected against these parameters.

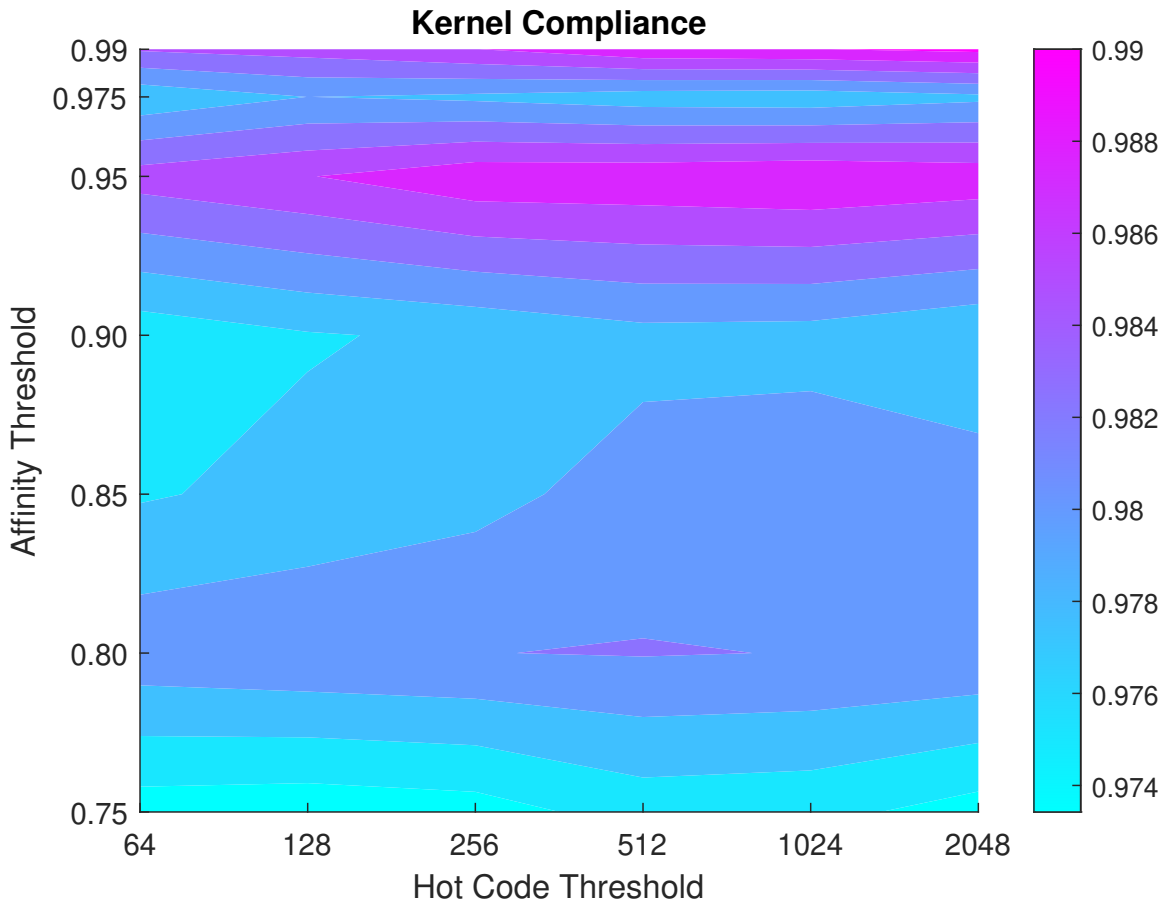


Figure 17: Kernel Compliance
The horizontal axis is the hot code threshold. The vertical axis is the affinity threshold. Compliance is whether the detected kernel conforms to the given definition. At an affinity of 0.8 and a hot code threshold of 512 a compliance level of 0.985 is achieved.

Within Figure 17 there is a local maximum at a hot code threshold of 512 and a threshold of 0.8. This is due to a bias in the input programs which were written to target 512 iterations. This can be seen as a higher point in Figure 18. Lower thresholds and higher hot code thresholds are detrimental to the number of kernels detected but will raise the quality of the kernels detected.

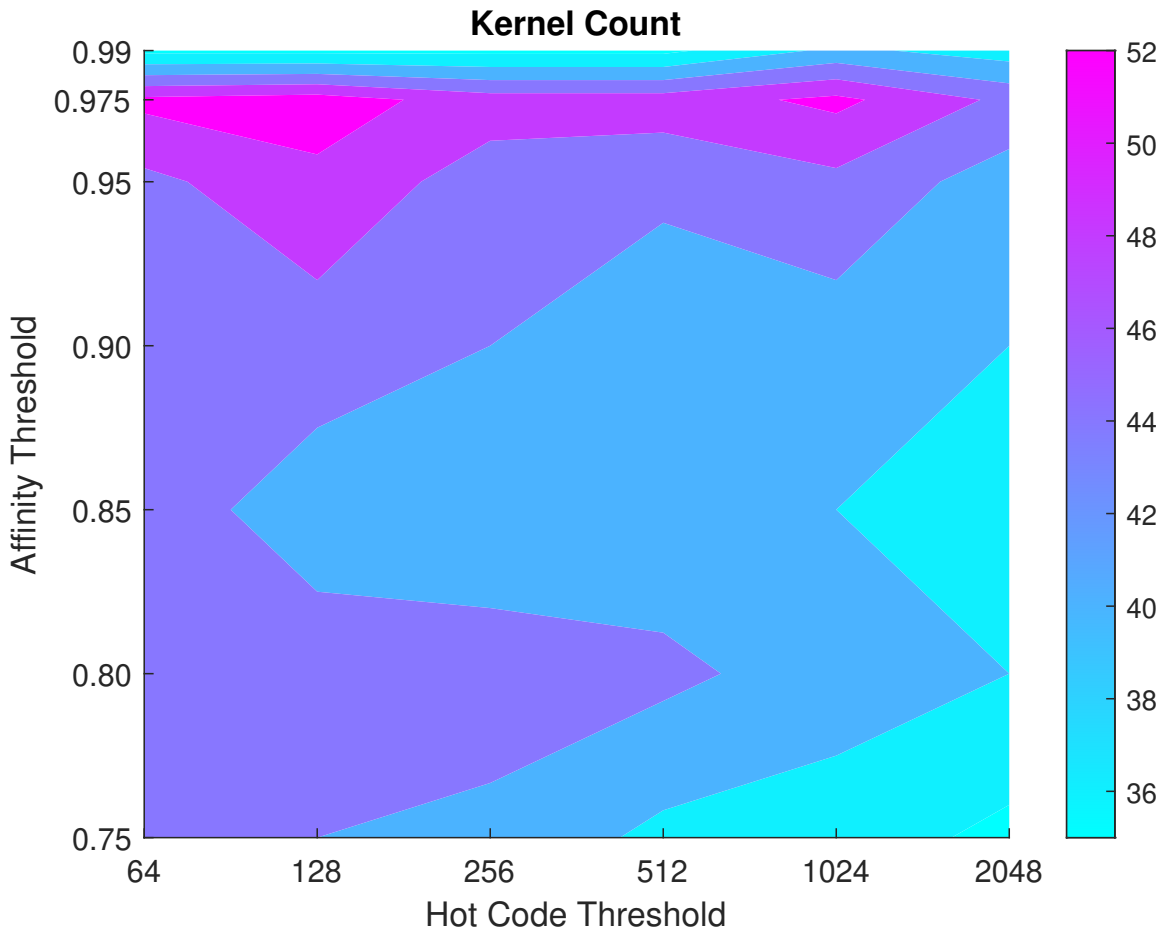


Figure 18: Kernel Count
The horizontal axis is the hot code threshold. The vertical axis is the affinity threshold. The elevation of this map is the median number of kernels detected per application.

3.6.3.1 Extraction Pipeline Kernel Effects

The blocks that compose a kernel change based on how far it has gone through the refinement process. Figure 19 illustrates the number of kernels detected in the top graph, the ratio of blocks explained in the bitcode in the second graph, and the ratio of blocks explained from the traces in the bottom graph.

The Heuristic Kernels are often the most prolific, but this number adjusts down from 60

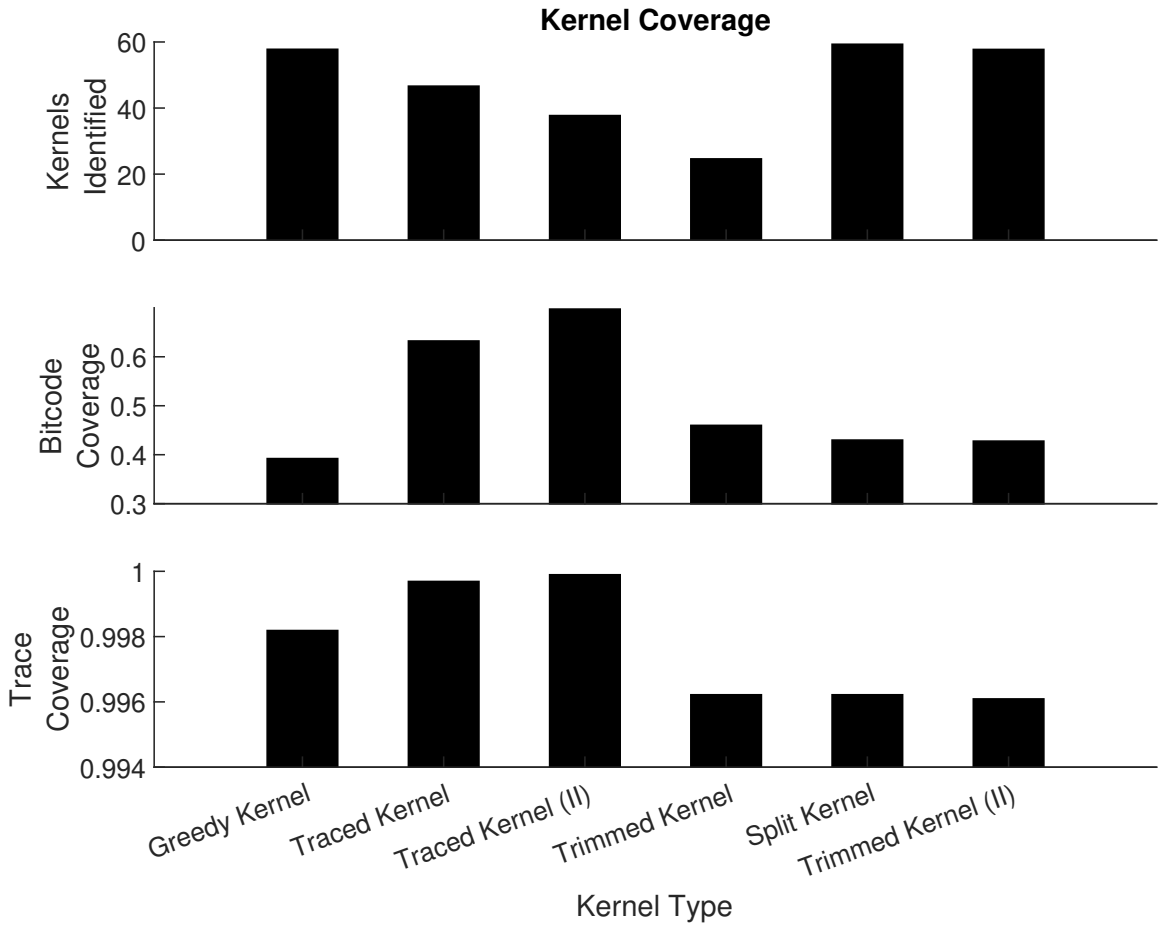


Figure 19: Detected Kernels Coverage
The top graph contains the median number of kernels detected from the input application set. The middle graph contains the median ratio of basic blocks in the bitcode explained by the kernels. The bottom graph contains the median ratio of basic blocks in the application trace explained by the kernels. The refinement process initially shrinks the number of kernels and increases code coverage. The Trimmed Kernel and Split Kernel detectors will then start removing blocks from the kernels, increasing the expressiveness of the kernels at the expense of a small amount of code coverage.

to 40 kernels as additional sub-branches are folded into the kernel by the first and second Traced Kernel passes. This process raises the number of blocks from the original bitcode that are explained by the kernels from 45% to 60%. This implies that on average 15% of basic blocks in a source application are a part of a kernel but are not themselves hot code.

The Trimmed Kernel detector then removes any non-recurring blocks from the beginning

and end of the kernels. This further lowers the number of kernels detected to 22 as duplicate kernels are dealiased. The removal of these blocks does lower both the number of blocks explained by these kernels in both the bitcode and trace; however, by removing these blocks, the kernels detected fit the kernel definition given in Section 3.3. Any blocks removed by this pass cannot recur and are not a part of a kernel.

The Split Kernel detector is responsible for splitting kernels that were merged erroneously. This dramatically increases the number of kernels detected. The number of blocks in the bitcode explained by the kernels decreases slightly due to the removal of non-recurring blocks, but the change in the trace coverage is virtually zero.

The final Trimmed Kernel detector is done primarily to make the kernels better conform to the definition. This slightly decreases the number of kernels detected, but the kernels removed were dealiased, not discarded.

The number of kernels detected is highly reliant upon the hot code threshold, where the hot code threshold is the minimum number of times a basic block must execute to be selected as a seed by the Heuristic Kernel detector. TraceAtlas uses a threshold of 512 by default, but a user should modify this value depending on the types of kernels they wish to detect.

3.6.3.2 Hot Code Threshold Effect

Figure 20 demonstrates the hot code threshold for a kernel to be detected. The data presented is prepared by executing TraceAtlas on the input programs with a hot code threshold of 64 and extrapolating the minimum count for a single block in a kernel to be at the minimum threshold. Of the kernels detected, 84% are still detectable at the higher ratio of 512. Failing to detect these kernels is not necessarily bad. By definition, these kernels

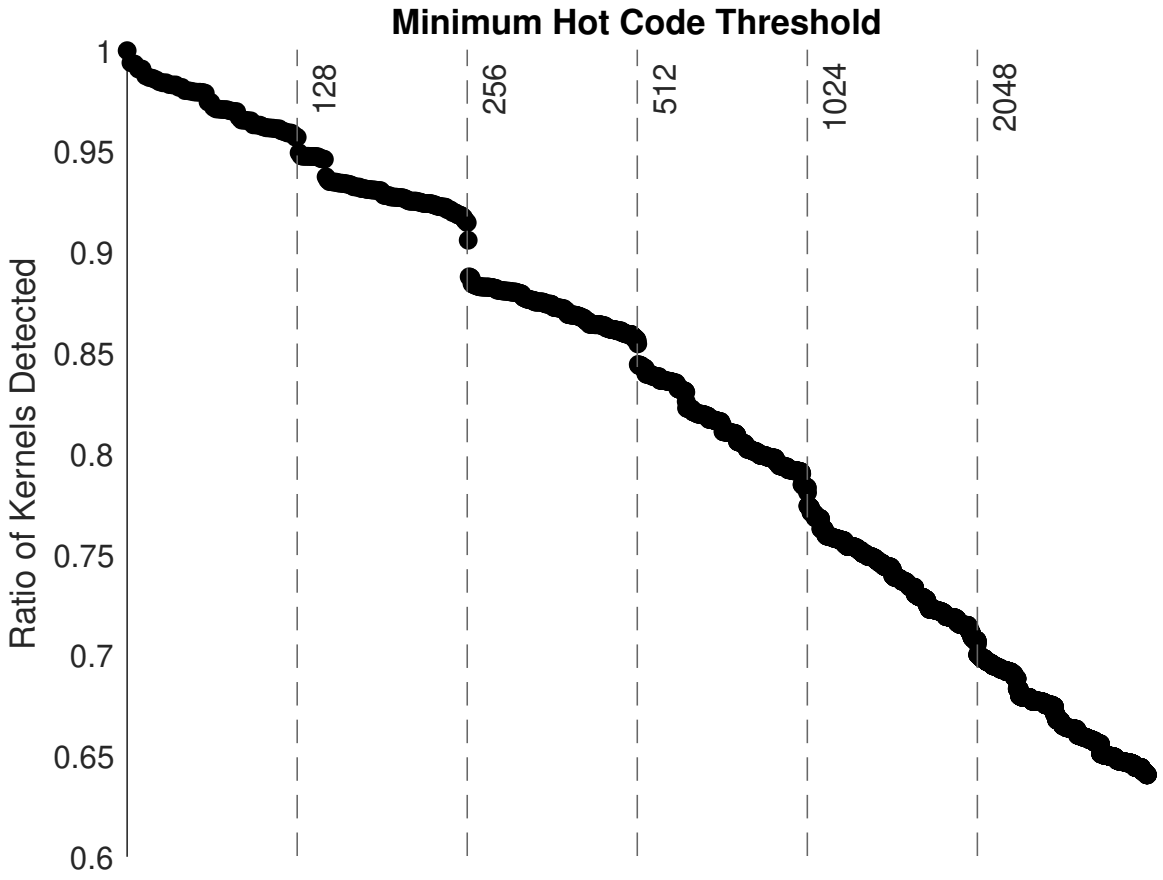


Figure 20: Minimum Hot code Threshold
The vertical axis are the ratio of kernels detectable. The horizontal axis is the minimum hot code threshold required to detect a kernel. As the threshold is raised fewer kernels are detected.

happen less than 512 times and thus cannot be a significant portion of the computation. Also, depending on the analysis in question an author may not care about these kernels, instead preferring to analyze those which compose a more significant portion of the computation.

There are a few jumps in this graph, the largest of which occurs at 128, 256, 512, and 1024. This is due to two factors. First, many of the applications in the test programs were written to conform to powers of two and thus will cluster on those values. Second, it is common practice in programming to write loops that conform to powers of two to maintain

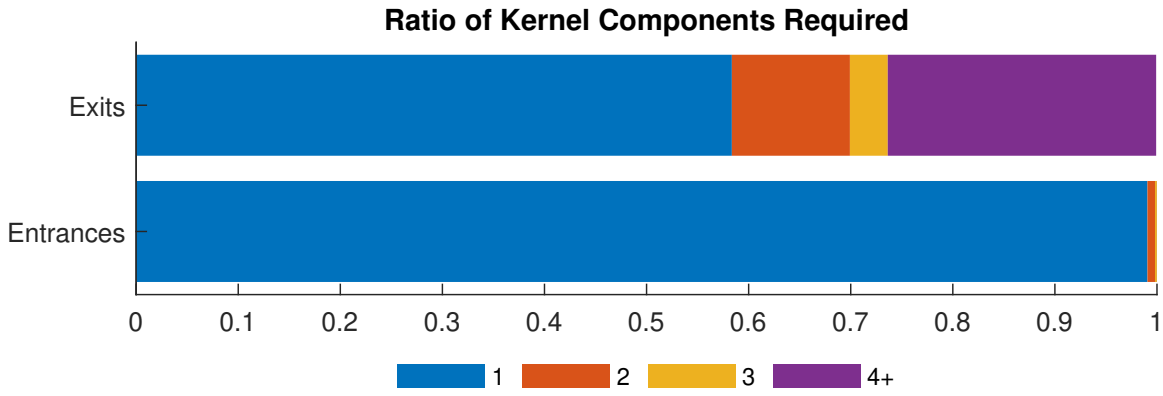


Figure 21: Kernel Entrance and Exit Count
In 99.07% cases, a kernel only requires a single Entrance. In 58.36% of cases a kernel only requires a single exit, but 26.29% of kernels detected require more than three exits.

byte alignment. Due to these facets of the code, a disproportionate number of kernels are executed on these edges and will have a corresponding minimum hot code threshold.

3.6.3.3 Required Kernel Properties

The kernel definition from Section 3.3 contains several attributes. A naive kernel that is compatible with traditional definitions will contain one entrance and one exit. Figure 21 demonstrates the number of Entrances and Exits required for kernels within the test application suite.

Over 99% of kernels contain a single entrance. This conforms to the traditional definition in the vast majority of cases; however, only 58.36% of kernels contain a single exit and a single conditional. This implies that most kernels from code in the wild break from their computations at various points, and it cannot be assumed that an entire iteration of the kernel will necessarily execute. To fully explain code in the wild, all of these attributes are necessary to represent kernels.

3.6.4 Summary

TraceAtlas generates dynamic traces that are 70% smaller than the current state-of-the-art while only having a time dilation factor of nine. It has been used successfully to extract 31,263 kernels from 1,821 applications. Each application contained a median of 42 kernels per application. We evaluated the kernel extraction algorithm with respect to the kernel affinity threshold and the hot code threshold. An affinity of 0.975 achieves the most robust results, while the hot code threshold had no significance to the robustness of the algorithm, but rather functioned as a filter to identify kernels the user is interested in.

These results demonstrate that generating a dynamic trace for kernel extraction can be done efficiently with a small disk and time footprint. The greedy algorithm previously discussed requires no user input and can identify kernels from dynamic traces with a small time overhead. Combining these techniques to identify kernels allows for wild code to be analyzed automatically, enabling the use of code parallelization tools on any program.

3.7 Discussion

Asanovic et. al has developed the most comprehensive kernel definition to date[7]. Their definition defines a kernel through enumeration and provides no way to identify kernels from source code, instead opting to require the user to express their code in this framework. There have been other kernel definitions that derive from Asanovic and fall victim to the same requirements[37]. DSLs define their kernels as a specific type of computation, but they also require that the code be written in a form they can understand. TraceAtlas allows for any kernel to be dynamically detected from source with no user interaction. Its definition is encompassing. TraceAtlas kernels are both compatible with all Asanovic kernels that

have a repeating computation as well as every DSL that expresses a kernel as a repeating computation.

TraceAtlas has a dependency upon LLVM IR being available for the source application. This limits the use of the tool to C and C++ projects with the potential for Fortran through the use of f18[63] or fc[64] or other languages with a custom LLVM frontend such as Julia and Rust. Supporting other interpretive[3] languages such as Python and Perl are possible if the control binary and supporting libraries are compiled with TraceAtlas injected. Closed source tools can still be used through an IR lifter such as LLVM-mctoll [65].

The traces produced by TraceAtlas are dynamic application traces, much like those from Aladdin[60]. As a result, they will only represent the paths taken in the execution. Unexplored kernels and dead code will not be traced or identified as a kernel in the resultant trace. Kernels from static code will contain these paths and the dead code. Care must be taken to ensure that the sample application executes the code of interest and that it executes a sufficient number of times given a user's hot code threshold. Although dynamic tracing is not without cost, TraceAtlas is able to create dynamic traces dramatically faster than current techniques. This has the potential to enable many code analysis techniques that have heretofore been too expensive to even consider on wild code.

3.8 Conclusion

Compute kernels are the fundamental abstraction used to identify speedup opportunities, structure programming languages, and design hardware accelerators. This work has demonstrated a technique for identifying kernels through the use of dynamic tracing. By

[3]Interpretive here refers to languages that require an external binary to execute. Compiled languages with library dependencies such as C# and Java do not apply.

using dynamic traces, it is possible to extract kernels from wild code. The application of this technique has no special requirements and can be applied to any application whose source code is available. Tracing code can be injected efficiently, and tracing only inserts a time dilation factor of nine. The resultant trace only produced one megabyte of data per second of trace. All of this makes potential dynamic trace tools an inexpensive and attractive solution to many problems.

This work also presents a log space algorithm that analyzes a dynamic trace to extract compute kernels from any application with available sources. This technique detects all the input kernels by the proposed definition, which encompasses many of the properties one expects in a kernel. This work evaluated programs from 23 libraries that contained 31,263 individual kernels instances. Automatic kernel detection from programs is possible with a small performance cost. Compute kernels can be extracted from any program without any user interaction at all.

Up until recently, working with an application trace was too expensive to perform any non-local analysis. The techniques presented in this work dramatically lower the cost of implementation, allowing for existing techniques to be more successfully applied and for new potential optimizations to be developed. Some potential applications of this work include polyhedral analysis[53], genetic algorithms[66], heterogenous code scheduling[67], kernel classification through ML inference[68], and direct compiler optimizations[69]. This work does not propose any actual optimizations, but it makes other optimizations possible.

3.9 Cross Library Validation

To perform some cross validation of the techniques described in this dissertation, four programs were analyzed close to the completion of the document. These programs use

Table 4: Final Cross-Validation Applications

Library	Application	Kernel Count
Aquila	AM-Modulation	9
Aquila	fft-filter	32
Aquila	mfcc-calculation	17
Crypto++	MD5	1

two different kernel libraries: Aquila for signal processing and Crypto++ for standard cryptographic operations. The applications presented were examples provided by the developers themselves. The applications analyzed and their kernel count are enumerated in Table 4. The applications were analyzed in depth and each of the kernels identified corresponded with expected cyclic behavior in the program.

This cross validation was performed to demonstrate that the kernel definition provided was not overfitted to the applications we were analyzing at the time. To this end, the applications demonstrated were not modified in any way. They were instead taken directly from the projects git repository or documentation and compiled to LLVM IR as any program can be. This was done simply by setting the compiler and linker flags to appropriate values. No other changes were required. A standard cmake invocation produced all of the required applications.

Due to the ease of analysis, roughly fifty new applications were ready for analysis from these two libraries; however, due to time constraints and complexity, only four of them were described here. Each of the given applications that were analyzed produced kernels that conformed to both our expectations of what a kernel should be and the proposed kernel definition.

The applications present in Aquila correspond directly to cyclic behavior explored in the application's execution. The simplest, AM-Modulation, was composed of nine kernels:

AM-Modulation (0), data destructor (1), data initializer (2), signal source generators (3 and 4), sine signal generator (5), a sampling function (6), and a plotting function (7 and 8 hierarchically). These kernels were validated by manually going through the source code to identify their function. mfcc-calculation and fft-filter both displayed similar behavior and were also analyzed manually. The larger number of kernels present is due to the more complex nature of the computations being performed.

Crypto++ was used as a final verification of more complex algorithms. It is a common library used for cryptographic operations in modern applications. To ensure functionality, a simple MD5 checksum was performed first. TraceAtlas correctly identified this as a single kernel.

This analysis is a small demonstration of the validity of the kernels identified by TraceAtlas. By analyzing the runtime behavior of the program in conjunction with the static code information it is possible to identify kernel as they are written with high fidelity.

Chapter 4

A REFINED, DECIDABLE KERNEL DEFINITION

A decidable kernel definition is necessary to algorithmically extract kernels from programs. Most modern attempts at kernel extraction derive from hot code[50] or from template matching. Both of these techniques are flawed and unable to represent all kernels present in the modern computational landscape.

Modern kernel optimization algorithms require that a kernel be preannotated. These algorithms make assumptions upon the format of the input code and should only be applied to compatible code segments. If the kernels required for these tools could be found automatically, it would be possible to apply these techniques to a larger cross section of programs.

The kernel definition presented in Chapter 3 has proven to be insufficient to detect all kernels in the sample application pool. As more programs were analyzed, edge cases started emerging that illustrated flaws assumption of how kernels loop. Merging runtime cycles has proven to be a complex task and a new definition was developed that marries the runtime information to the static source code. Section 4.1 goes into proper depth on this new kernel definition.

The kernel definition has been tailored and refined to contain no dependence on the dynamic trace. This allows for an algorithm to be developed that does not need to know about the full execution behavior. Instead, an algorithm can operate solely on the local branching behavior which is far more dense in information. Additionally, this approach is more in line with modern techniques and requires no trace at all. Section 4.3 describes a sample algorithm that can successfully extract kernels with minimal runtime information.

This algorithm does not handle all edge cases, but instead serves as a demonstration that the kernel definition both works and can be utilized to find kernels more efficiently. The algorithm as written allows for kernels to be detected in any program and has no known limitations.

4.1 Kernel Definition Constraints

A decidable kernel definition allows for the automatic extraction of kernels from wild code. To make this definition useful in the real world, it must satisfy two constraints: it must be decidable (return a true or false answer), and it must be within $\mathcal{P}$ (the set of algorithms that execute in polynomial time). Much like how NP-Complete problems have a polynomial verifier, a useful kernel definition must also possess one. The true complexity of identifying kernels within a program is currently unknown, but it is suspected to be NP-Complete. This work does not have a proof and thus makes no authoritative claim to the complexity of the problem. This definition is quick to calculate and deterministic for any code collection. This makes it ideal for kernel extraction.

The proposed kernel definition is built upon four constraints that define inclusion in the kernel group. These constraints are all designed to be verifiable in polynomial time. First, the kernel graph must recur (Section 4.1.1). Second, a kernel must be distinct of other kernels (Section 4.1.2). Third, a kernel must be more likely to recur than to terminate (Section 4.2). Fourth, the blocks two hierarchical kernels do not share must be able to start from outside the kernel (Section 4.2.1).

These constraints are applied to the modified CFG utilized in Chapter 3. This CFG varies from traditional CFGs by inserting function calls into the computational graph as regular edges. Similarly, function returns and exceptions also insert an edge back to the

call site. This transformation is similar to inlining without code duplication. The CFG is also extended to possess runtime information by attaching the number of times a branch was executed at runtime to each edge. This information can be acquired through compiler annotations or a modified version of the tracing algorithm in Chapter 3. This algorithm would only increment a value for each edge rather than storing the entire computational history.

The branching behavior of individual basic blocks can be built up by constructing an adjacency matrix during runtime. Just like with the prior dynamic tracing technique, it is possible to call code every time a block is entered. By simply incrementing a counter that corresponds to each basic block pair, the values in this matrix can be constructed with minimal runtime overhead. It is important to note that it is ill advised to store the entire matrix. A single adjacency matrix would compose n^2 values that would cache poorly in memory. The vast majority of values in this matrix remain zero due to that section of the graph not being explored. A simple hash map is a much more efficient storage mechanism that should always be utilized. During testing, the generation of this data was found to have a non-measurable runtime overhead. The only measurable time overhead was the exporting of the final matrix to the disk. Timing measurements of the runtime without exporting the final values demonstrated no change in runtime.

This graphical representation with the aforementioned adjacency matrix contains all details required to test the four constraints. The static program information is encoded as simple edges. The runtime program information is stored as weights on the edges which represent the total number of times that path was taken through the computation. This provides all the information necessary to extract kernels from the graph with the new definition.

4.1.1 Constraint 1: A Kernel is a Strongly Connected Subgraph

Every basic block within a kernel must have the opportunity to perform a recurrent computation. This pattern forms a loop and is represented as a cycle within the graph. This behavior can be represented as a strongly connected subgraph, a subgraph where there is a path between all nodes in the subgraph. This implies that there is a possible computational path that will execute every other part of the kernel. Effectively, these blocks are a part of the same loop. Any singular block that is not a part of the strongly connected subgraph cannot be a part of the kernel because it is impossible for it to create a recurrent computation. Basically, if a block is not a part of the strongly connected component it cannot "loop". See Figure 22 for an example. Path ABC forms a loop, but block D has no path back and thus cannot be a part of the kernel.

A kernel is not necessarily a strongly connected component. Strongly connected components differ in that they must by definition be maximal. Strongly connected subgraphs can overlap partially or entirely. Since loops can nest within each other, kernels also must be able to exhibit this behavior. The top-level kernel is almost always a strongly connected component, but all child kernels are simply strongly connected subgraphs that are subgraphs of the top level kernel.

This constraint can be checking quickly in polynomial time. The naive approach would be so check if there is a path between every two nodes in the kernel. There are at most n^2 node combinations to check. Dijkstra would take at most $E + n^2$ operations to determine this[70]. Thus, the upper bound for the computation is n^4 if the most naive algorithm is utilized. This demonstrates that this constraint satisfies $\mathcal{P}$. There are more optimal solutions to this problem, but this dissertation is only interested in demonstrating that it is possible to solve for these constraints quickly. Because of this, all further constraints will only be

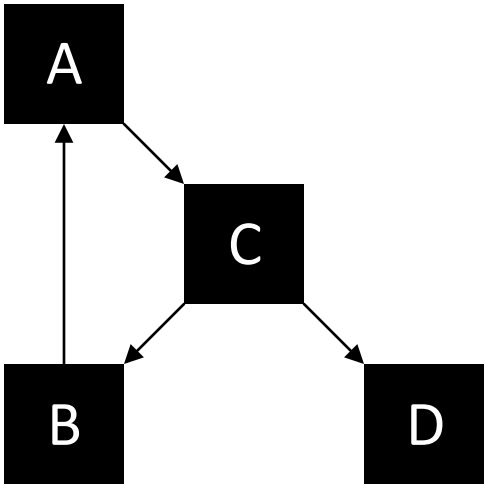


Figure 22: Constraint 1 Example
Blocks A, B, and C form a strongly connected graph. The addition of block D would make the graph not strongly connected. Thus the collection of A, B, C, and D cannot be a kernel by Constraint 1.

checked for membership in $\mathcal{P}$. The given algorithms for checking these parameters are not optimal, but they both validate membership and are quickly decidable. A variant used in this work is visible in Figure 23.

```
for node a in graph:
    for node b in graph:
        if len(Dijkstra(a, b)) == 0:
            return False
return True
```

Figure 23: Constraint 1 Pseudocode
Constraint 1 can be checked in polynomial time by simply checking if there is a path between every two nodes in the graph. Dijkstra takes at most $E + N\log(N)$ operations[70]. This would happen N^2 times giving an ultimate computational complexity of $N^2 * (E + N\log(N))$ which is within polynomial time.

This constraint only requires that every kernel be a part of a loop. It is possible that a kernel's subgraph will overlap with other kernel subgraphs. Conditional logic specifically will introduce multiple strongly connected subgraphs within a single loop. To address this, a new constraint that bans partial overlaps requires that a kernel be complete.

4.1.2 Constraint 2: Every Kernel Must Not Partially Overlap With Any Other Kernel

Requiring kernels to not overlap compensates for conditional logic by forbidding kernels that only share some blocks. Strongly connected subgraphs overlap in three primary scenarios. First, they can be nested kernels where one loop exists within another. This scenario would have the parent kernel as a superset of the child kernel; thus, nested loops are in conformance with this rule. Figure 24a is an example of this. Blocks B, C, and D are in both loops and are thus shared between kernels. The entirety of the child loop is contained by the parent loop and is thus in conformance with the rule.

The second manner in which strongly connected subgraphs can overlap is through conditional logic. Such graphs can possess conditional logic that creates multiple cycles through the graph. Figure 24b is an example of this. Path $A \rightarrow B \rightarrow D$ is a strongly connected subgraph but so is path $A \rightarrow C \rightarrow D$. These two smaller subgraphs fail the constraint due to sharing A and D but not sharing B and C. The correct kernel in this case would contain all four blocks.

The final way kernels can overlap in this graph can be through a shared function call. Figure 24c is an example of this where blocks A, F, and B form a kernel, and blocks C, F, and D form a kernel. F is a shared function call between the two loops. For the purposes of classification, function calls are treated as if they were inlined. By checking for this

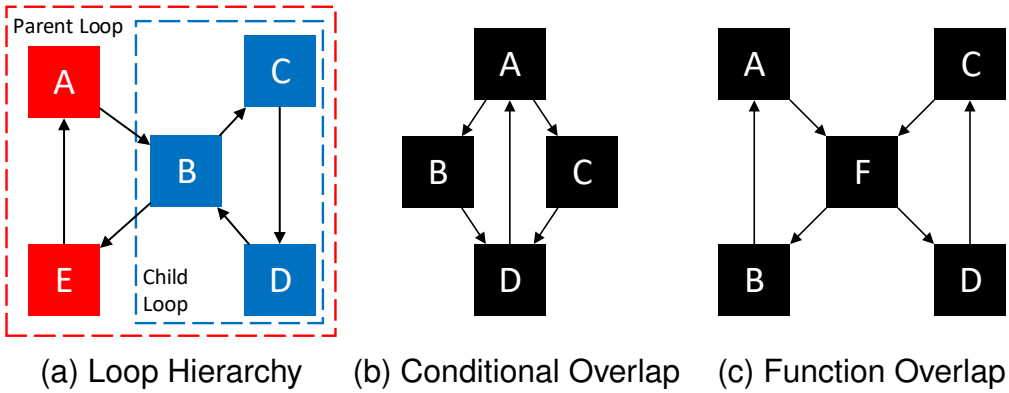


Figure 24: Strongly Connected Subgraphs Overlap
It is possible for strongly connected subgraphs to overlap in three manners. They can be a part of a loop hierarchy and form a valid kernel (a). They can be only a section of a conditional block and be an invalid kernel (b). They can be composed of a function overlap and a valid kernel (c). There is no other way for strongly connected subgraphs to overlap.

constraint, it is guaranteed that when a kernel is identified, the entire kernel will be extracted. No low frequency block will be left out.

This constraint can be checked in polynomial time by performing a basic set intersection between every pair of kernels. There are at most k^2 combinations of kernels (where $k << n$). It would take at most n iterations across a pair of kernels to check if the blocks fully overlap, partially overlap, or do not overlap. As a result this verifier belongs to $\mathcal{P}$. Pseudocode of the algorithm used in this work is available in Figure 25.

If two kernels are found to be discordant as is the case in the two smaller cycles in figure 24b, this constraint does not address which kernel is correct, if either. It only claims what is and is not a kernel.

The combination of Constraints 1 and 2 require that loops within code form kernels, but they make no mention of runtime behavior. Kernels in the real world actually loop many times and dominate the runtime of the input program. Although a loop may exist in code, it

```
for node a in kernelA:
    if not a in kernelB:
        aSuper = True
    else:
        share = True
for node b in kernelB:
    if not b in kernelA
        bSuper = True
    else:
        share = True
if aSuper and bSuper: # graphs are coincident
    return True
elif not aSuper and not bSuper:
    # graphs at most partially overlap
    if share:
        return False
    else: # graphs do not overlap at all
        return True
else: # graphs are hierarchical
    return True
```

Figure 25: Constraint 2 Pseudocode

Constraint 2 can be checked in polynomial time by simply checking how the nodes in the kernels overlap. If the graphs fully overlap (first if statement) they are the same kernel. If the partially overlap (second if statement) it is illegal. If they are hierarchical it is legal. The two for loops at the beginning each take n^2 time. Thus, this constraint can also be checked in polynomial time.

may rarely execute or never exhibit looping behavior. To make this a requirement, a new constraint is required.

4.2 Constraint 3: A Kernel Must be More Likely to Recur Than to Terminate

The core principal of a kernel is that it recurs. This recurrent execution results in kernels dominating the execution. Simple computations that ideal kernels form make a clean cycle

which can be easily observed both graphically and in the trace; unfortunately, real programs have branches, function calls, exceptions, and other structures that make such analysis more difficult. The core principal that allows for kernel discrimination is that a kernel is always more probable to continue looping than to terminate.

If any individual branch is more likely to exit than to continue recurring, it behaves like linear code and thus is not part of the kernel. An example of this would be a sorting step in an algorithm that only operates on sorted data. Since it is always sorted, it only executes once and is not a kernel in the computation. Figure 26 demonstrates a graph that would be a kernel by Constraints 1 and 2. This figure also labels the probability of taking a branch in the computation. From block C, it is more likely to take the branch to D. This indicates that the path to B is rare and cannot represent a significant portion of the computation. As a result there is no kernel within this figure.

This property can be checked in polynomial time. To do so, simply check the successor of each node to ensure that the probability of taking a path that stays within the strongly connected subgraph is more likely than the probability of leaving the subgraph. This algorithm executes in $O(nb)$ where n is the number of blocks in the kernel and b is the average number of neighbors per block. Being linear, this check also satisfies membership in $\mathcal{P}$. Pseudocode of the algorithm used in this work is available in Figure 27.

This check still makes it possible for an algorithm to exclude a low probability branch which is present in a parent kernel. These branches are rare executions, but they are a part of the loop and must be represented. The most obvious examples of this are boundary conditions in image processing. When an algorithm reaches the edge of an image, it operates with a different value than that provided by the image. Since this path can only be entered

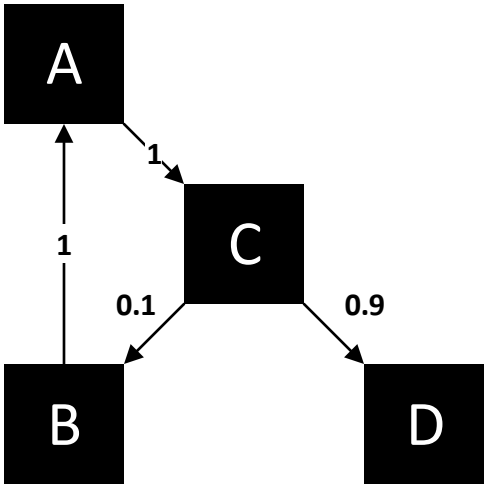


Figure 26: Probable Path Example
By constraints 1 and 2, blocks A, B, and C should form a kernel. The edge weights refer to the probability that the computation will take that path. From block C there is a 0.9 chance that it will choose to take path D and behave like a linear code segment, skipping the loop with block B. Block B is a rare scenario and cannot therefore represent a significant portion of the computation. By constraint 3, there is no kernel in this figure.

through the kernel code, any kernel non-overlap must possess at least one edge that can enter the cycle.

4.2.1 Constraint 4: The Difference Between Two Nested Kernels Must Contain at Least One Incoming Edge

When two kernels form a hierarchy, the set of blocks in the parent kernel that are not in the child is the logic that happens in between iterations of the child kernel. When thinking of kernels as for-loops, this section forms the conditional, the incrementor, and additional glue logic. The glue logic can be used to manipulate memory or perform further tasks, although this is less common. In general, the primary purpose of this section is control flow.

```
for node in kernel:
    probIn = 0
    probOut = 0
    for successor in node:
        if not successor in kernel:
            probOut += successorWeight
        else:
            probIn += successorWeight
    if probIn < probOut:
        return False
return True
```

Figure 27: Constraint 3 Pseudocode

Constraint 3 can be checked in polynomial time by checking that each node is more likely to remain within the kernel than to exit. For every node, sum up both the probabilities of leaving the kernel and the probabilities of staying in the kernel. If the probability of leaving is greater than the probability of staying in, this kernel is in violation of Constraint 3. This constraint can be checked within $E * N$ time where E is the average number of edges from a node and N is the total number of nodes. This falls within linear time with respect to the size of the graph. Thus, it is within polynomial time as well.

Compilers generate hierarchical loops with a placeholder block between them. Because of this, there must be a path through the computation that enters this section without entering the child kernel first. If the path enters the child kernel first, there is nothing that distinguishes this section from an early exit (exceptions/break statements/goto). Since this is a strongly connected subgraph due to Constraint 1, an inability to enter this segment without entering the child kernel also implies that this section also reenters the child kernel. This makes the entire structure behave like a single loop, and it would be more concisely described as a single kernel.

Figure 24b is the most common scenario that causes this behavior. If branch B is 100x more likely than C, block C could be excluded by constraints 1-3. Thus two kernels would be formed from these blocks, ABD and ABCD. Since the only path through kernel ABCD that isn't in kernel ABD is a simple conditional from ACD, it doesn't refer to a hierarchy.

To represent the conditional correctly, both branches must be combined. This would only be a valid hierarchy if there was an entrance to block C, creating a separate looping possibility and thus a separate kernel.

Although a graph can be constructed that breaks this rule, there is no known programmatic way to construct one that conforms to our understanding of kernels. The simplest way to do this is through prolific use of goto statements, but graphs of this construction no longer resemble loops but instead cliques and thus should be excluded. Optimization by the compiler can also create graphs that break this rule, but this usually happens as a part of loop optimization. The resultant "loop" now has been effectively fused with the parent kernel, forming just a single kernel.

This constraint can also be tested in polynomial time through the use of basic set intersections. To test, compare every two kernels (n^2), and for every block that is not shared, check if there is a path from outside the kernel to this section that does not go through the nested kernel (at most b^2). This puts the check within $\mathcal{P}$. Note, it is advised, but not required, that this be the last check performed to ensure that the only overlapping kernels are hierarchical. Pseudocode of the algorithm used in this work is available in Figure 28

These four constraints are adequate to ensure that all kernels identified are valid kernels. These constraints do not require that all kernels be identified. They assume that all relevant computations have been identified in advance. The checks to verify that these kernels have been included do not necessarily fall within $\mathcal{NP}$. As a result, this work also advises two optimizing principals be utilized to guarantee all kernels are fully represented.

4.2.2 Optimizing Principal 1: Every Block Belonging to a Cycle Must be a Part of a Kernel

Requiring that every cycle in the CFG refer to a kernel (or attempt to form a kernel) guarantees that all kernels are found and that all branches in kernels are fused. This is technically unnecessary if the given algorithm can properly distinguish desired kernels from undesired kernels from low probability branches. Without further information, it is safe to assume that all cycles are a kernel.

Only loops that recur will form a cycle in this modified CFG. Due to the runtime information present, cycles will only occur if the code recurred. The cycles that are executed only once or not at all will fall foul of Constraint 3 and be discarded. This makes it a safe check all scenarios.

This can be tested for in polynomial time through a simple graph exploration. For every block in the graph, simply utilize Dijkstra to explore the graph to find a path to itself. If it does find such a path, is it part of a cycle and should be part of a kernel. Since Dijkstra belongs to $\mathcal{P}$, so does this verifier.

This principal does not indicate if the given kernel is correct but rather if all kernels in the input program are explained. Essentially all basic blocks that are not within a kernel are deemed to be unimportant. It is strongly advised that all cycles are a part of a kernel to guarantee all paths through the execution are included. Depending on the application, this may or may not be desired behavior, so it is not a part of the definition.

This optimization principal guarantees that all relevant basic blocks are contained in at least one kernel. The prior constraints and this principal do not ensure that hierarchical kernels are found. To ensure that nested kernels are identified, it is desirable that a basic block form the smallest possible kernel that also conforms to the above constraints.

4.2.3 Optimization Principal 2: Every Kernel is of Minimal Size Such That All Constraints are True

To guarantee all nested kernels are detected, it is desirable for all kernels to be minimal so long as the above constraints are satisfied. Nested kernels can be technically ignored, and all the above constraints are legal. This may be the desired behavior depending on the intended application of the kernels.

To the knowledge of the author, this rule cannot be tested without combinatorics. Testing for it would imply iteratively constructing kernels with smaller subgraphs. This algorithm requires testing each subgraph combination and thus does not belong to $\mathcal{P}$.

Essentially, this rule defines the types of kernels we would like to find, not what they are. This changes the verification problem to be an optimization problem which is far more difficult to solve. This is akin to the difference between NP-Complete algorithms and NP-Hard algorithms. An algorithm can avoid testing for this principal by building kernels from the smallest components first, but there will not be a guarantee that all nested kernels are represented.

4.2.4 Summary of Kernel Properties

Kernels in this model are dependent upon runtime behavior and program composition. This contrasts with the definition utilized in Chapter 3 which only checked if code could recur. The bubble sort algorithm was identified as two separate kernels, but it is only one by these constraints. Bubble sort is composed of two loops and was identified as two kernels by the prior algorithm. Upon analysis with these constraints, the resort step is rare, resulting in the resort step being a false positive and not marked as a kernel. Examples like these

were identified as a part of the refinement process, demonstrating that the constraints are performing the task as designed.

Ultimately, the question of what a kernel is today depends on an individual user's opinion on required kernel properties. The rules presented here are a combination of what the author believes to be important to the structure of a kernel and additional constraints to exclude programs that do not conform to expected "kernel" behavior. These constraints are both broad and decidable. By being broad, other kernel definitions can be described. By being decidable it is possible to write algorithms to detect the kernels.

Traditionally, hypotheses like this are tested by providing a contradictory example, but there is no current formal definition of a kernel with which to work. This definition is the first attempt to create such a definition and thus cannot be compared yet. To disprove this definition, it is necessary to identify or manufacture a non-kernel graph that would satisfy these constraints. Conversely, an example of a non-kernel that exhibits the behaviors expected of a kernel could also indicate a flaw with these restrictions.

To form a baseline comparison, the kernels identified through these constraints were compared with those that were identified by the kernel extraction algorithm detailed in Chapter 3. Initially there were fewer rules which resulted in either far more or far fewer kernels depending on the stage of development. The kernels were initially flawed, leading to the constraints and optimizing principals being refined. Today, all kernels identified by the algorithm in Chapter 3 are now explainable through these rules. The few exceptions which did not conform to these constraints were manually affirmed to not be kernels.

This definition is not perfect or final. As the definition is utilized more, false positives and false negatives will likely be identified that will require updating the constraints. The definition as presented can find all kernels present in the final database used in this work.

The kernels come from over sixteen libraries, including nearly four hundred programs and over ten thousand kernels.

4.3 Refined Kernel Detection Algorithm

To demonstrate the utility of the kernel constraints, a prototype greedy kernel extraction algorithm has been developed. It is still under active development and does not support shared function calls. Similarly, it uses the naive checks for every constraint and thus is relatively slow. It does guarantee the prior four constraints are satisfied as well as Optimization Principle One. Optimization Principle Two is likely to be satisfied, but it is not guaranteed since this algorithm is greedy.

The algorithm is broken up into two phases. The first phase identifies seed kernels by identifying all blocks that can recur. This is derived from Optimization Principle One. The second phase legalizes kernels by iteratively fusing or rejecting kernels. Figure 29 contains the pseudo-code for this algorithm, written in Python.

This algorithm creates seed kernels from the maximal likelihood cycle. This cycle represents the most likely route to be taken in a loop that contains the seed block. It starts by creating a probability transform of the graph. This probabilistic transform changes the edges to be positive with the smallest weights referring to the most probable paths. This enables use of simple pathfinding algorithms like Dijkstra. These cycles are then iteratively fused if they are illegal to form legal kernels.

The graph produced by TraceAtlas is transformed into the more convenient form with a simple arithmetic transformation. The original weights in the graph are the raw number of times a particular branch is selected. First, every edge is transformed into a probability by dividing the weight by the sum of all outgoing weights from a node. Then, it is again

transformed by taking the negative log of the probability. This guarantees that all paths are positive values, with lower weights implying more likely routes. Figure 30 demonstrates how these weights would change in a sample CFG. The weights from $A \rightarrow C$ and from $B \rightarrow A$ are 1 and become 0 because that path is guaranteed. The path from $C \rightarrow D$ is probable at 0.9. The negative log of 0.9 is 0.0457. Since the value is small, it is probable. The path from $C \rightarrow B$ is less likely and results in a larger value. The negative log of 0.1 is 1 in this example.

With the probability graph prepared, the most probable cycle can be identified for every block by executing Dijkstra on every node to find a path to itself. This will efficiently find the most probable cycle for every basic block. Duplicate cycles can be avoided by storing them in a simple set structure. Although this does introduce a cost of storing the cycles, it is an important optimization for larger programs.

The first step in legalizing the cycles is to fuse them with the most similar cycle. This is determined by the descending ranking of similarities between the kernel graphs. This is measured by dividing the sum of the weights of the shared edges of the graphs from the non-shared edges. This portion is highly greedy and not guaranteed to be optimal. The question of how to measure graph similarity is still an open research question. This heuristic is an adequate measure, but other sorting mechanisms would also work.

The algorithm does not check if kernels partially overlap at this stage. The cycles are changing rapidly, and a partial overlap does not yet have significance. Fusing partial kernel overlaps here will result in fusing too aggressively, discarding smaller kernels. This heuristic does not guarantee kernels are only fusing from the inside out, so any extra fusing will obscure smaller kernels. Once the kernels are legal save for partial overlaps, any remaining partial overlaps can be fused together. This guarantees that any overlaps refer to alternate paths through the computation.

Finally, there is an edge case where a kernel can be composed of only other kernels. This algorithm will not identify such a kernel due to the identified cycles already having maximal likelihood. To adjust for this, each kernel should be replaced with a single node and the algorithm executed again. If there are no new kernels found, then the algorithm can terminate since it has found all kernels. This is a structure that is difficult to write in C, but technically possible. Figure 31 demonstrates what this code would look like. Blocks A, B, C, and D all form kernels in their own right. The combination of them also forms another kernel. Without this step, the algorithm would not detect the parent kernel due to there being no intermediate blocks that aren't already explained by another kernel.

4.4 Summary of Updated Kernel Definition

A kernel can be checked for validity by four constraints and two optimizing principals:

1. Constraint: A Kernel is a Strongly Connected Subgraph
2. Constraint: Every Kernel Must Not Partially Overlap With Any Other Kernel
3. Constraint: A Kernel Must be More Likely to Recur Than to Terminate
4. Constraint: The Difference Between Two Nested Kernels Must Contain at Least One Incoming Edge
5. Principal: Every Block Belonging to a Cycle Must be a Part of a Kernel
6. Principal: Every Kernel is of Minimal Size Such That All Constraints are True

This proposed kernel definition conforms to the authors' belief in a kernel's definition. The resultant kernels are cyclic, encompass conditionals, compatible with any code form, and able to represent kernels represented by other works. This definition will allow for

algorithms that identify kernels programmatically for the purposes of better optimization and cross compilation.

A simple greedy algorithm can be utilized to find kernels efficiently without a significant runtime overhead. The illustrated algorithm is not optimal and relies upon heuristics, but it identifies kernels without user interaction.

```
nonShared = set()
predecessors = set()
# find all non-shared nodes
for a in kernelA:
    if not a in kernelB:
        nonShared.insert(a)

# check for entrances
for node in nonShared:
    toVisit = predecessors(node)
    visited.add(n)
    for n in toVisit:
        if n in kernelB:
        toVisit.erase(n)
    # toVisit has only predecessors for n not in kernelB
    visited = set()
    found = False
    while len(toVisit) != 0:
        n = toVisit.pop()
        if not n in kernelA and not n in kernelB:
            break # found an external kernel entrance
        elif not n in kernelB:
            for p in predecessor(n):
                if not p in visited:
                    toVisit.add(p)
    # failed to find an entrance
    if found = False:
        return False

return True
```

Figure 28: Constraint 4 Pseudocode

Constraint 4 can be checked in polynomial time by checking if there is an entrance for every difference between hierarchical kernels. Start by finding the nodes not shared in a hierarchy (n^2). Then perform BFS to identify if there is a path into the kernel that doesn't enter the nested kernel (b^2[71]). The upper boundary of the complexity of this algorithm is $n^2 + b^2$. Thus, it is within polynomial time as well.

```
def findKernels(graph):
    priorResult = set()
    while True:
        paths = set()
        for block in kernel:
            path = Dijkstra(block, block)
            if path != None:
                paths.add(path)

        finalResult = set()
        for path in paths:
            while True:
                if not IsLegal(path):
                    score = 0
                    mostSimilar = None
                    for comp in paths:
                        sim = getSimilarity(path, comp)
                        if sim > score:
                            mostSimilar = comp
                            score = sim
                    path += mostSimilar
                else:
                    finalResult.add(path)
                    break

        graph.CollapseKernels(finalResult)
        priorCount = len(priorResult)
        for kernel in finalResult:
            priorResult.add(kernel)
        if priorCount == len(priorResult):
            break
    return priorResult
```

Figure 29: Kernel Algorithm Pseudocode

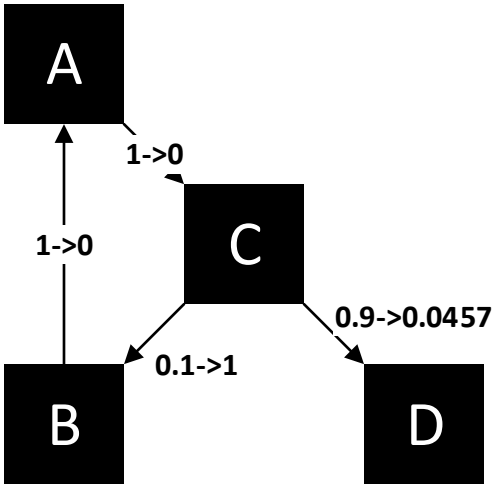


Figure 30: CFG Weight Transformation

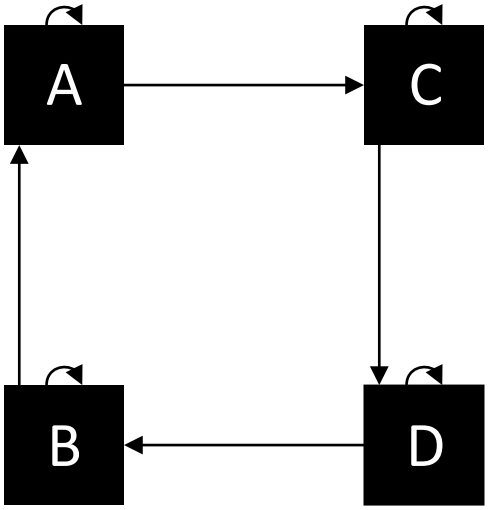


Figure 31: Pure Nested Kernel Example

Chapter 5

PREDICTING KERNEL LABELS FROM STATIC AND DYNAMIC PROGRAM INTRINSICS

To meet ever increasing performance metrics, modern software systems leverage accelerators to run code faster and more efficiently. System architects optimize these accelerators for a particular kernel structure to achieve significant performance gains. These accelerator optimizations prove detrimental or fatal to the execution if a kernel does not conform to the prescribed code structure; unfortunately, compilers struggle to infer accelerator compatibility, instead relying upon the user to specify how it should execute. To make code in the wild compatible with these accelerators, the first step is to detect which kernel label they belong into.

To predict a kernel's label, we developed a series of machine learning models which take in statically inferable and dynamically measurable attributes about kernels. Using these features, the machine learning models in this work accurately predict the kernel labels applied by the application authors. Using more traditional models such as SVM, accuracy approaches 80% while newer models like DNN achieve greater than 96.8% accuracy. The final DNN design robustly predicts FFT kernel labels, even on code structures it has never seen before. This demonstrates that the prediction of a kernel's label is possible on wild code using only static and dynamic program intrinsics.

5.1 Introduction

Advancing process technologies no longer provide performance improvements to new generations of processors. Instead, system architects design modern processors to either prioritize scalar-performance or energy efficiency[72]. Additionally, thermal constraints push systems to specialize areas of a chip[50]. The resulting heterogenous systems contain several specialized accelerators, a Domain Specific System on Chip (DSSoC).

Current compiler technologies cannot determine which accelerator a kernel should execute on without either human interaction or manual labeling. Currently, developers manually label code with compiler macros[73] or domain specific languages[8][52][74]. An automated method of detecting the kernels present and identifying the category of said kernels will allow for the compilation of wild code to high-efficiency accelerators with minimal user interaction.

The kernel annotation is fundamentally important due to the complexity of compiling to heterogenous systems. Each accelerator has a different compilation flow, code compatibility, and overhead costs. Compiling to such a system requires the developer be both familiar with the trade-offs with all the architectures on the SoC and fully understand the intricacies of how their code executes[30]. This expertise is expensive and limits the adoption of such a system to a few, performance critical applications.

Modern accelerators often rely upon a Domain Specific Language (DSL) to simplify the compilation and optimization of kernel code. For example, Ragan-Kelley et al. developed the Halide language to simplify the design of stencil computations which operate most efficiently on the GPU[8]. Software architects continue to develop compilers to specifically optimize kernels, but the complexity of applications often limits these compilers to one domain. The TVM compiler from Che et al. enables automated code optimizations for

machine learning kernels running on a variety of CPUs and DSSoCs, but it does not support other kernel families[75]. Similarly, developers utilize stochastic super optimization [76] and evolutionary computation-based code optimization[77] to automatically extract performance gains for general-purpose CPU or GPU kernels. All these works depend on having small, focused kernels presented to them. Wild kernels are often large and written inefficiently. Before these tools can operate upon the kernels, either programmers or another tool must pre-annotate the code.

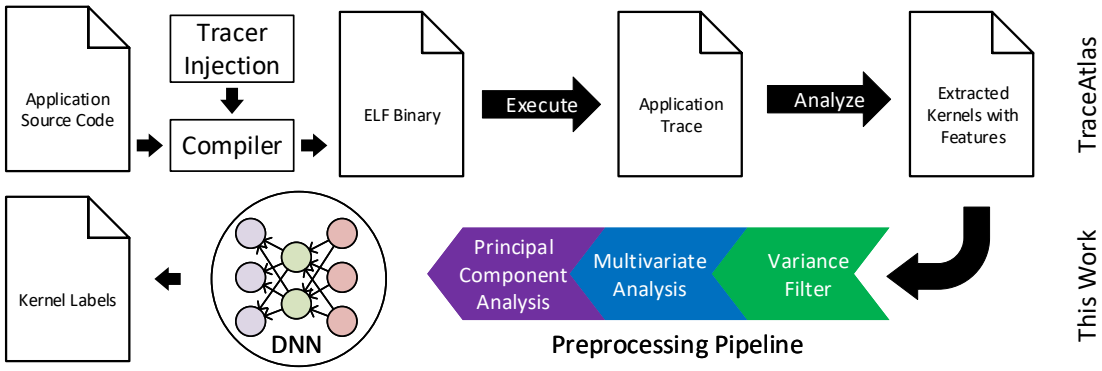


Figure 32: TraceAtlas Analysis Pipeline

Extracting kernels and their labels from wild code requires a suite of tools, executed in order. First, TraceAtlas generates a dynamic trace from the original source code by injecting a tracer and running the resultant executable natively. Second, TraceAtlas analyzes the trace to extract the kernels based on the kernel behavior. Both steps are not described in this work. Afterwards, the kernels are available for inference. To improve the fidelity of later tools a preprocessing pipeline transforms the kernel metrics first. The pipeline filters features by variance, utilizes multivariate analysis to determine which features are most important, and finally perform Principal Component Analysis to reduce the dimensionality. A DNN then uses the data to predict the kernel label. Developers can then use these labels to aid compilers or to schedule code on a heterogenous system.

Accelerator specialization makes kernel classification closely correlate with the architecture of best fit. Detecting that a kernel is an FFT directly implies that it should execute on an FFT accelerator. Stencil computations should operate on the GPU. BLAS kernels should target a linear algebra accelerator. Predicting this label allows the compiler to directly

specify which architectures will execute a kernel most efficiently without requiring the user to provide any information.

This work proposes a machine learning (ML) pipeline to predict a kernel's label (see Figure 32). First, TraceAtlas gathers the kernel data[30]. Second, the ML pipeline preprocesses the data by filtering the variance, applying multivariate analysis, and performing Principal Component Analysis. This work takes a principled approach to identify LLVM IR features that are most relevant to the proposed machine. Finally, the pipeline trains the classifier against this data to predict a kernel's label from wild, unlabeled code.

This work evaluated the proposed learning-based approaches for computational kernel labeling with a large code base of 1477 different programs containing 8 different kernel labels. The four classifiers (logistic regression, Support Vector Machines, K-Nearest-Neighbor, and Dense Neural Networks) studied in this work achieve 75.0%, 86.1%, 93.1% and 96.8% accuracy, showing promising potential as an accurate, automated kernel labeling tool.

This paper contributes:

- A method to extract features from input programs for the purposes of machine learning
- A statistically robust domain reduction of the input feature space
- A Dense Neural Network that predicts a kernel's label with high accuracy

5.2 Background

Predicting kernel behavior requires both a concrete kernel definition and the ability to extrapolate program behavior. Kernel definitions vary significantly between domains, but this work will operate on the TraceAtlas kernel definition due to it encompassing most other kernel definitions and allowing for automated extraction. The utilization of static and

dynamic attributes from kernel behavior provides information that is valuable for system optimization. It allows for the prediction of performance across architectures, aids in ASIC design, and assists in heterogenous system scheduling. With these features, it is possible to use machine learning to both predict kernel performance across a broad swath of architectures as well as aid in actual compiler optimizations[78].

5.2.1 Computational Kernel Definitions

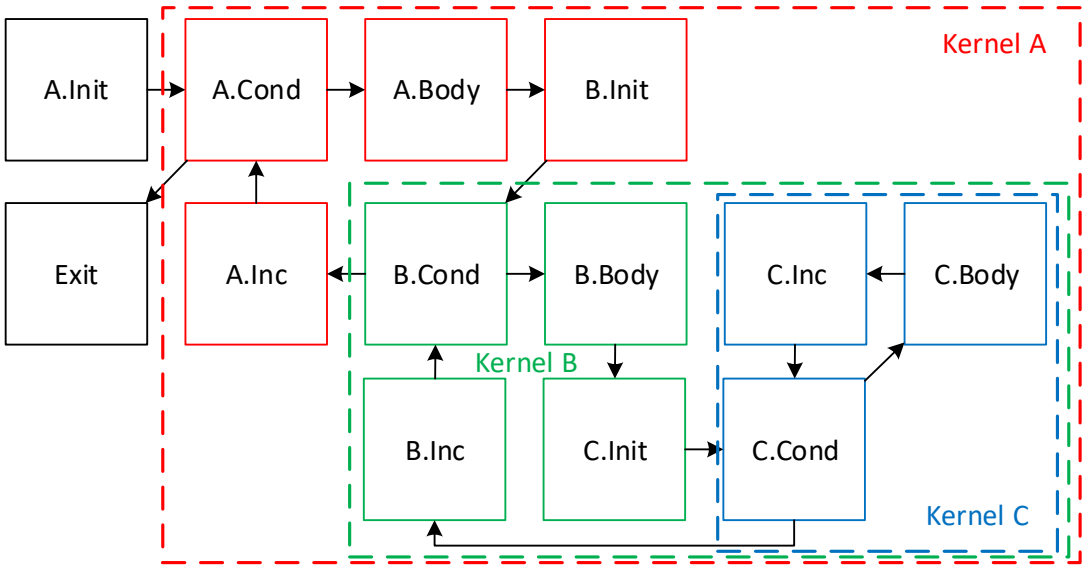
This work utilizes the TraceAtlas definition of a kernel[30]. This definition utilizes an abstraction that transforms an application into a directed graph, allowing for program analysis without the user providing any information. Within this graph, every node represents a basic block, and every directed edge represents a succession from one block to the next. This graph is analogous to a Control Flow Graph. Specifically, TraceAtlas defines a kernel as a strongly connected subgraph within this graph. TraceAtlas places additional constraints on the kernel to guarantee the graph includes all the control flow. This definition allows for the extraction of kernels from any program if the source code is available. This system accepts any program, dramatically lowering the cost of adding additional samples to a collection of kernels.

The TraceAtlas kernel definition encompasses all loops as well as recursion and any other programmatic behavior that results in a control flow recurrence. Figure 33a is a matrix multiply "kernel" that encompasses three different loops and therefore three different kernels. Figure 33b is an example of the successor graph where each kernel includes only the blocks that can reach every other block within the kernel. This definition makes kernels "loop" and dominate the execution of the input program. TraceAtlas's decidable kernel definition makes them a convenient platform for analyzing kernel labels in wild code.

```
int* MultiplyMatrices(int* A, int* B, int d0, int d1, int d2) {
     int* result = malloc(sizeof(int)*d0*d2);
     for (int i = 0 ; i < d0 ; i++ ) {
          for (int j = 0 ; j < d1 ; j++ ) {
               int sum = 0;
               for (int k = 0 ; k < d2 ; k++ ) {
                    sum += A[i][k]*B[k][j];
               }
               result[i * d0 + j] = sum;
          }
     }
     return result;
}
```

Kernel C
Kernel B
Kernel A

(a) Matrix Multiply Annotated Code



(b) Matrix Multiply DD-Path

Figure 33: Matrix Multiply Kernel Example

Nested loops result in a hierarchy of kernels. The naïve form of Matrix multiply contains three nested loops resulting in a hierarchy of three kernels (see 33a). When the compiler converts the code to a Decision-to-Decision Path (DD-Path), the kernel hierarchy becomes clear (see 33b). The parent kernel (Kernel A) fully contains the child kernels (Kernel B and C).

Asanovic et. al proposed thirteen kernel categories which explain a “diverse set of important applications”[7]. Their kernels have no formal definition, but they are instead an

enumeration of a series of classes of kernels. This list does not necessarily encompass all kernels. This work selected these labels to represent important code patterns. This definition does not define a kernel by its properties, but instead relies upon categorization. To collate and compile thousands of kernels with their definition requires hand labeling every kernel, a laborious task. Automatically labeling kernels requires a property-based definition, such as the one proposed by TraceAtlas.

Hashimoto et. al. conducted a survey of Fortran kernel computations on GitHub and categorized them as a subset of Asanovic's kernels[37]. They identified six kernel types in the input programs. To categorize them, they laid the kernels out on a decision tree for ease of identification. Their work only looked at this subset of kernels due to them working with Fortran, a language known for expressing mathematical kernels. This work looks at a broader set of computation as they exist in the wild. Identifying kernels from all code requires the ability to represent all kernels. Fortran, specializing in mathematical kernels, cannot express many common kernel types. The source kernels in this work will instead come from C and C++ codes, expressing both mathematical kernels like in Hashimoto's work, as well as more pedantic kernels as many developers often write.

Every computational domain develops their own set of Domain Specific Languages that simplify writing software at the expense of broader compatibility. Each DSL that expresses a kernel utilizes an abstraction that makes implicit assumptions specific to their field. Halide defines a kernel as a combination of a call schedule and a core function[8]. Gramps defines each kernel as a stage that pushes data into buffers for inter-kernel communication[52]. Aspen defines a kernel as a core computational function with a parallelism attribute[74]. All these DSLs are kernel specializations which rely on a core kernel abstraction with additional limitations. Fundamentally, a DSL possesses a recurrence relationship with additional constraints on the memory access pattern, the loop pattern, or both. Despite this shared core

abstraction, there is no current method to transform between DSLs. This work takes the perspective that requiring a programmer to learn and write a DSL is prohibitively expensive. To take full advantage of these disparate hardware platforms, a compiler must efficiently exploit the program and architecture without manual intervention.

5.2.2 Extrapolating Program Behavior

Program behavior in one environment is indicative of behavior in other, dissimilar environments. Developers use kernel performance on a CPU to predict performance on other platforms such as GPUs[79, 80], Bit.Little cores[81] or full SoCs[82]. This indicates that the way a program executes in the CPU models the performance on other platforms. Extending on this, a better understanding of the state of kernels on the CPU should transfer over to other domains.

Developers use many sources to extract features from programs. Shao and Brooks developed a set of architecture independent features[83]. Aladdin selects features to demonstrate primarily memory and control behavior; thus, they only have seven different features. A key concept in their features is the entropy of branches and memory accesses. This work takes the view that the types of instructions executed are more important than branching behavior. The quality of written code will significantly perturb the memory footprint and control flow. The types of instructions executed are intrinsic in the type of computation performed and cannot vary significantly. Hoste and Eeckhout developed a set of architecture independent features[84]. Many of their features also derive from performance metrics, including instruction level parallelism, register traffic, working set size, and data stream strides. This work determined that any exclusively performance-based feature does not adequately represent the data set.

5.3 Extracting Features from Programs

Programs are complex systems which developers model with low level Intermediate Representations (IR). Factors such as other executing programs, the OS, the hardware platform, and anything else executing at the time can impact the behavior of code, but the IR simplifies a program to its structure, excluding any runtime information. This work extracts ML features from high-level LLVM IR, which is predominantly platform agnostic. By analyzing the source program in isolation, it is possible to make inferences based exclusively on the original application.

This work annotates kernels within LLVM IR (Section 5.3.1), describes how to generate the features used in future ML models (Section 5.3.2), and enumerates the source libraries and applications used to train the models (Section 5.3.3).

5.3.1 Structure of Features and Kernels

TraceAtlas formats kernels as a collection of basic blocks containing LLVM IR and extracts kernels from application traces. To provide expert information on the kernels, this work adds an API to TraceAtlas to inject labels into the trace. A label is inserted into the trace through a function call, a "KernelStart" or "KernelStop". This labels all kernels between these annotations as a member of that label. Figure 34 demonstrates how a developer would insert labels into source code. Because TraceAtlas defines kernels at a lower level than most authors think about kernels, the labels do not imply a kernel is that label. Instead, a labeled kernel implies that the kernel contributes to that label, not that the kernel is said label. TraceAtlas kernels posses kernel hierarchy. For example, three kernels together compose Matrix Multiply: the inner, outer, and middle loops. The

```
void kernel(float* a, float* n) {
  KernelStart(1DConv);
  for(int i = 1; i < 1023; i++) {
    n[i-1] = (a[i-1] + a[i] + a[i+1]) / 3;
  }
  KernelStop(1DConv);
}
```

Figure 34: Kernel Labeling Example
A developer can label every kernel through a call to a "KernelStart" and "KernelStop" function as shown with a C API.

Table 5: Descriptions of Kernel Labels

Label	Description	Count
Correlator	Relationship between vectors	42
FFT	Fast Fourier Transform	1955
FIR	Finite Impulse Response	363
IIR	Infinite Impulse Response	342
MatrixMultiply	Matrix multiplication	236
RandInit	Data initialization kernel	513
ZIP	Element-wise vector kernel	7
Transpose	Matrix transpose	458

collection of kernels is the full Matrix Multiply while any individual kernel may simply provide supportive functionality.

Individual domains develop their own unique sets of labels that are important to their field. For this work, the code contains nine labels that are important to radio-processing applications (Table 5). The labels selected are not representative of all the kernels in the input data set.

5.3.2 Features in Programs

This work generates features by sampling the number of times a type/instruction pair occurs in a kernel's LLVM IR. LLVM implements 64 different op codes. The number of datatypes in LLVM is technically unlimited due to the existence of structs and other compound types, but there are a fixed number of base data types enumerated: void, float, integer, array, vector, and pointer. This work labels any non-primitive type as "other" for the purposes of feature extraction. Overall, the combination of op codes and data types produce 920 unique features. Table 7 enumerates a subsection of kernel features that are the most valuable for kernel labeling.

Gathering features from both the dynamic trace and the original bitcode provide more holistic information about the program. This work extracts features from bitcode by counting the total number of times a particular combination is present in the static LLVM IR and the dynamic trace.

The raw features collected from the trace are highly susceptible to subtle program changes. The features are raw counts of the number of times a particular instruction/type combination occurs. By executing a kernel loop twice as much, the features will vary dramatically.

Normalizing the input features by the total instruction count of the kernel makes them comparable. With this modification, a change in the loop count will only slightly change the ratio. This is because the dominant blocks in the kernel execute the same number of times relative to each other. Similarly, normalizing the bitcode features by instruction again maintains the relationship between instruction types while diminishing the impact of loop-unrolling and other optimizations.

Table 6: TraceAtlas Test Corpus

Library	Apps	Kernels	Version
StreamIt[10]	16	137	1.6
Halide[8]	41	977	2019/08/27
Gnu Scientific[11]	264	3,157	2.6
Perfect[12]	51	264	1.0.0
FEC[13]	8	167	3.0.1
MiBench[14]	27	170	1.0
SHOC[15]	10	35	1.1.4
VOLK	57	129	2.0.0
FFTW[16]	84	4,686	3.3.8
Dhrystone and Whetstone[17]	7	74	2
OpenCV[18]	74	6,827	4.1.0
mbedTLS[19]	70	1,902	2.16.3
Cortex Suite[20]	19	702	2019/10/30
Eigen[21]	47	2,958	3.3.6
FFmpeg[62]	9	66	4.2
spuce[22]	93	849	0.4.3
LiquidSDR[23]	114	1,409	1.3.1
Armadillo[24]	184	1,496	9.600.6
SigPack	56	2,243	1.2.4
In-House	590	3,015	
Total	1,821	31,263	

5.3.3 Input Programs in Kernel Collection

The applications present in the data set represent many computational domains. They span matrix operations, radio processing, encryption, image processing, and computational benchmarks. In total, the training data contains over 31,000 kernels available from over 1800 unique traces. This work traced and analyzed all the input programs with TraceAtlas v0.3.2[4]. An overview of the most prolific libraries present in the data set is available in Table 6.

[4] Available at https://github.com/ruhrie/TraceAtlas

The applications tested are a combination of in-house applications, drivers for libraries, and benchmarks. This work only modifies benchmarks by inserting kernel labels. This work compiles each library to a static library containing LLVM IR with no modifications.

5.3.4 Summary

This work extracts distinct input features from input kernels using TraceAtlas and analyzes over 9000 kernels from 1477 different applications. These features originate from the static source code (bitcode) and the dynamic application trace. This broad swath of data is adequate to train the ML models.

5.4 Domain Reduction

Machine learning models often struggle to converge with high dimensional, non-uniform datasets due to the curse of dimensionality. TraceAtlas generates 920 distinct input features from every input kernel. A preprocessing pipeline transforms the input features to remove extraneous features and lower its dimensionality. Figure 35 demonstrates the processing pipeline. The pipeline first filters by the variance present to remove features that never change in the input. Then, the pipeline applies multivariate analysis to only include features necessary for the model. Finally, the pipeline applies Principal Component Analysis (PCA) to combine correlated features and further shrink the number of output variables. The ML models then consume the result of this preprocessing pipeline.

A preprocessing pipeline (Figure 35) combines several techniques to efficiently reduce the dimensionality of the data. With the kernels gathered from TraceAtlas (in red), Variance Filtering (in green, see Section 5.4.1), Multivariate Analysis (in blue, see Section 5.4.2),

and Principal Component Analysis (in purple, see Section 5.4.3) successfully reduce the dimensionality from 920 to 32 with a minimal loss of entropy in the input data.

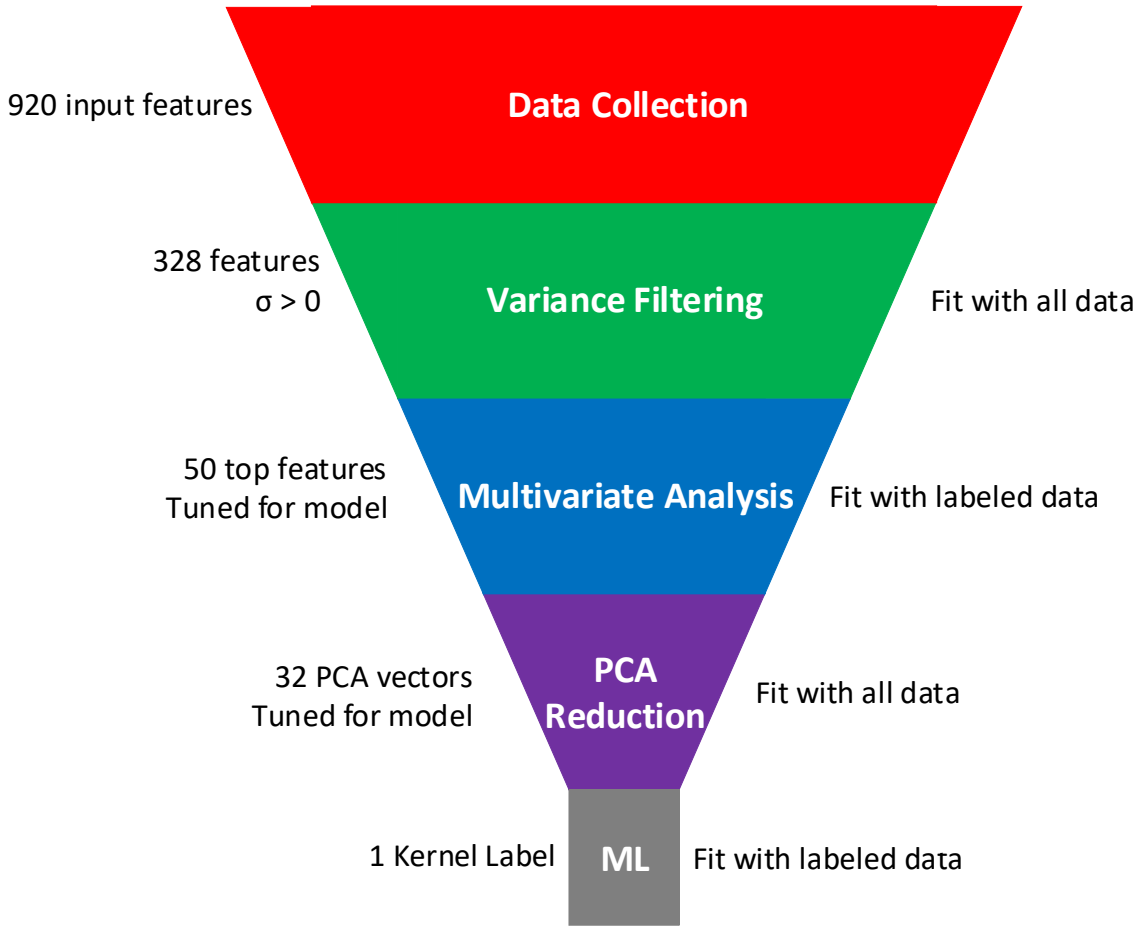


Figure 35: Data Feature Engineering Pipeline
Sample data begins as 920 input features gathered from both bitcode and dynamic traces. The pipeline then filters the data by removing any feature with a variance of zero. Afterwards, the pipeline applies multivariate analysis by selecting the top fifty features from their mutual information scores based upon labeled data. Finally, the pipeline applies PCA to shrink dimensionality of the data to 32 features based upon all data. This processing pipeline reduced the number of features from 920 to 32 with no detrimental effect to any of the machine learning predictors in this work.

5.4.1 Variance Filtering

TraceAtlas produces 920 distinct features to represent the code. Some of these features are logical impossibilities (such as an add that returns void). Others are rare combinations that do not exist in the data set. By analyzing the variance of these features, the pipeline discards variables which do not contribute to the machine learning models, shrinking the solution space. The first step of the pipeline is Variance Filtering (depicted in green in Figure 35).

A feature is significant if it has a non-zero variance. A variance of zero indicates the data never changes and is irrelevant. This work analyzes the variance with all the kernels available, both labeled and unlabeled. Ultimately, only 328 features have a non-zero variance. It is possible that some of these features will contain data in the future with new programs that produce a perturbation of this variable, but with the current dataset, they are unnecessary.

5.4.2 Multivariate Analysis

This work performs Multivariate Feature Analysis to identify the most important input vectors for prediction and further reduce the dimensionality. This is the second stage of the pipeline, shown in blue in Figure 35. This work utilizes scikit-learn to calculate the mutual information score with the algorithm described by Ross[85]. The mutual information score for each feature utilizes only the labeled data due to necessity of having a label to extract information with this algorithm.

Figure 36 demonstrates the mutual information provided by each factor. There is a dramatic drop that occurs at 50 features, indicating that features beyond 50 are significantly

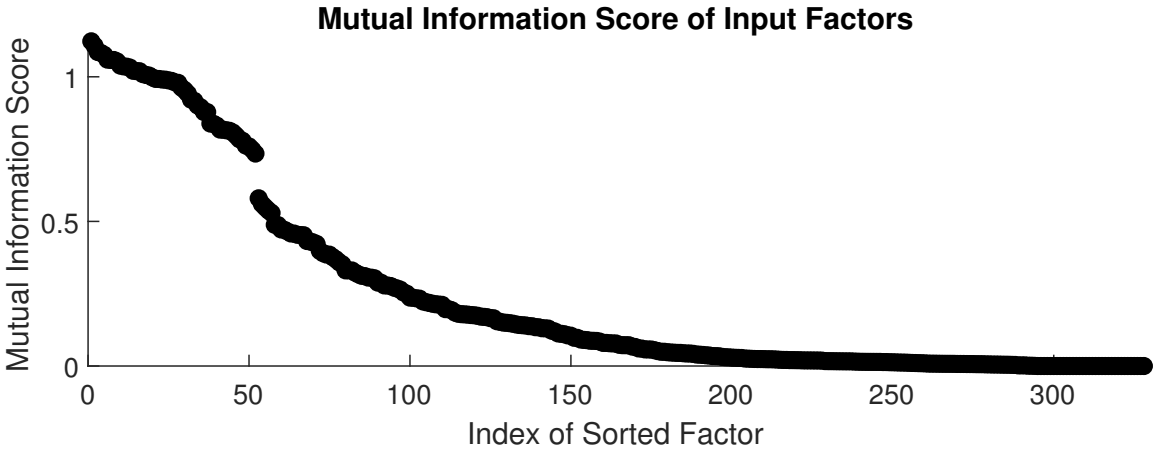


Figure 36: Multivariate Mutual Information Score
The mutual information added by each subsequent component drops as the feature count increases. The y-axis is the mutual information score, and the x-axis is the rank of each factor after sorting in descending order. The first twenty features are valuable and have a larger slope. The next thirty prove to also be valuable. After the top fifty features, there is a drop in the mutual information score, indicating that the pipeline should use the top fifty components.

less valuable. Excluding these features will lose a minimal amount of information. The final ML models later verify this in Section 5.5.3.

Tables 7 and 8 enumerate the selected features in decreasing order of mutual information score. The source specifies where the feature originates: the bitcode or the trace. The name is a brief description of the feature selected. Each feature is a count of instructions with a specific LLVM op code, instructions which return a particular type, or instructions of a particular LLVM op code and a particular type. Multivariate analysis selects features from both the trace and the bitcode in equal measure, indicating that both provide useful information to the models. Similarly, the features selected from the trace and the bitcode roughly correlate.

The information provided by the trace is invaluable in predicting the kernel labels. Of the top 20 features, 80% of them originate in the trace. The information provided by the

Table 7: Top 20 Selected Multivariate Features

Source	Operation	Type	Score
Trace	load	*	0.226718955
Trace	store	void	0.225048457
Trace	store	*	0.224025260
Trace	*	int	0.223063173
Trace	icmp	*	0.217838149
Bitcode	load	*	0.217071187
Trace	icmp	int	0.216411017
Trace	br	*	0.208979381
Trace	br	void	0.208335399
Trace	*	void	0.207211087
Trace	load	pointer	0.205307637
Trace	load	int	0.204529774
Trace	*	pointer	0.204263811
Bitcode	*	int	0.204814671
Trace	add	*	0.203326643
Trace	add	int	0.202250825
Trace	getelementptr	*	0.198506295
Trace	getelementptr	pointer	0.198121228
Bitcode	store	void	0.191746287
Bitcode	store	*	0.191082905

bitcode, although less significant, is still necessary to achieve the high level of precision in the models. Overall, the bitcode represents 63% of the remaining features. Prior work prioritized the utilization of static features. Tables 7 and 8 demonstrates that by combining these two metrics, models can make inferences more easily than with either by themselves.

5.4.3 Principal Component Analysis

Principal component analysis (PCA) combines features based upon entropy to further lower the dimensionality. This is the final step of the pipeline and is the Principal Component Analysis stage (shown in purple) in Figure 35. The features selected by multivariate analysis often derive from similar features. With this information, the models train against a smaller

Table 8: Next 30 Selected Multivariate Features

Source	Operation	Type	Score
Bitcode	*	pointer	0.189267632
Bitcode	*	void	0.188760057
Bitcode	load	pointer	0.188691750
Bitcode	load	int	0.185826714
Bitcode	icmp	*	0.180582732
Bitcode	icmp	int	0.179609901
Bitcode	br	*	0.178612148
Bitcode	br	void	0.176430869
Bitcode	add	*	0.169266180
Bitcode	getelementptr	*	0.169090562
Bitcode	getelementptr	pointer	0.168195618
Bitcode	add	int	0.166374617
Trace	phi	*	0.148018871
Trace	alloca	*	0.140032125
Bitcode	phi	*	0.139645516
Trace	ret	*	0.139496521
Trace	alloca	pointer	0.138546036
Bitcode	alloca	*	0.138277972
Trace	ret	void	0.138661705
Bitcode	alloca	pointer	0.136606860
Trace	call	*	0.133666062
Trace	*	Float	0.132981443
Bitcode	call	*	0.132100117
Bitcode	*	typeFloat	0.131920452
Trace	bitcast	*	0.131872297
Bitcode	ret	*	0.131846541
Trace	load	float	0.131338298
Bitcode	ret	void	0.131180710
Trace	bitcast	pointer	0.130736721
Trace	call	pointer	0.128240767

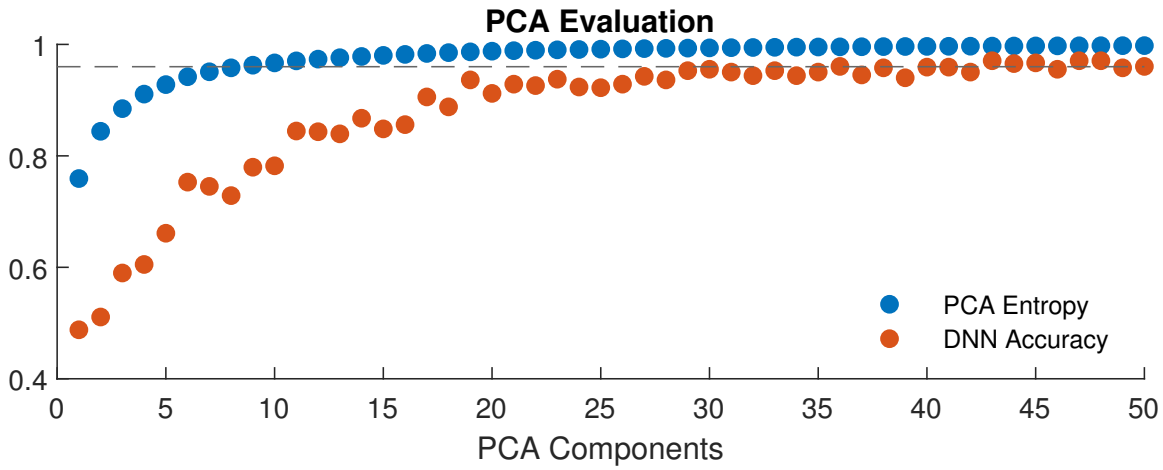


Figure 37: Accumulated PCA Entropy
The horizontal axis refers to the total number of selected PCA components. The vertical axis refers to the ratio explained or the DNN accuracy. This work trained each DNN 11 times with 5000 epochs to converge. The graph reports the median accuracy.Although the model only requires 22 features to explain 99% of the entropy, the accuracy of the DNN does not converge until the model utilizes 32 features. The final small ratio of entropy in these PCA components, although insignificant numerically, contribute to a high accuracy predictor. The dashed line represents the accuracy level at which the DNN converges, 96.8%.

feature space with a similar amount of entropy. This work trained the PCA model against the entire data set. Later machine learning models utilize only the labeled data. The output of this final PCA representation is the input for all the ML models.

A model typically selects a PCA vector count against an arbitrary threshold, like 90% of the entropy; the data set in this work only requires the first four components to explain over 90% of the entropy. For a higher threshold of 99%, the model only requires 22 features. Figure 37 demonstrates the accumulated entropy of the first 50 PCA components. Alternatively, many use the knee of the plot to select the vector count. This occurs at approximately 10 components representing 97% of the entropy. Based on this behavior at most 22 components should be necessary by conventional metrics. Features that explain a small ratio of the entropy can be necessary for robust classification. Upon analysis of

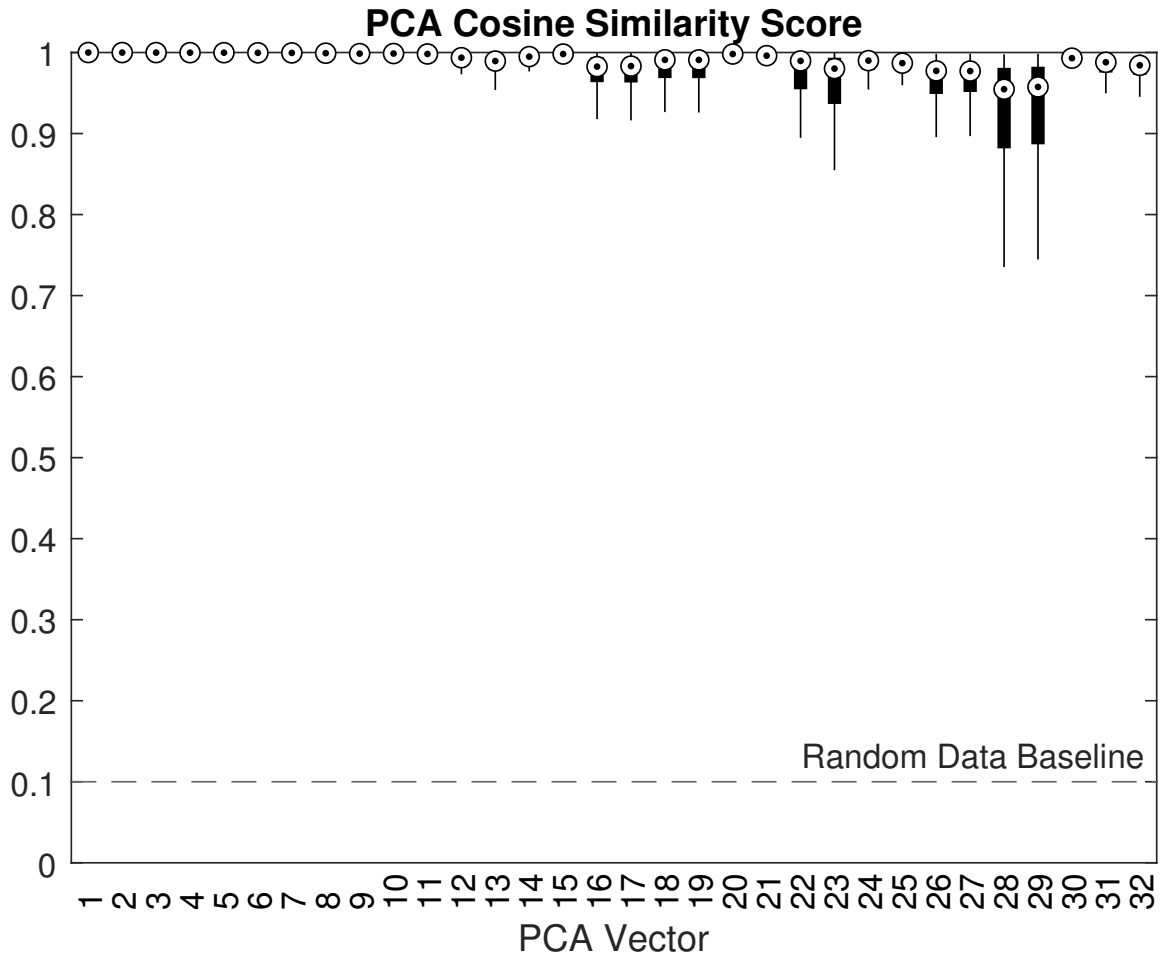


Figure 38: PCA Cosine Similarity
This work evaluates the PCA vectors by splitting them in half 500 times and comparing them to the original vectors through cosine similarity. The y-axis shows the magnitude of the cosine similarity for a vector after the Gale-Shapley algorithm matches them. The x-axis shows each PCA vector index. This whisker plot demonstrates that 23 of the PCA vectors are extremely stable. The remaining nine vectors vary more but are still consistent with the original features. This demonstrates that the PCA vectors are broadly representative of the entire dataset.

the final DNN model (Section 5.5.3), the pipeline needs to maintain at least 32 vectors to maximize the accuracy.

The extracted PCA vectors are stable. To validate their stability, this work splits the kernel suite in half 500 times to generate new PCA vectors and compares each of these

vectors to the original PCA vectors with their cosine similarity score. To match the vectors should they change order, this work utilizes the Gale-Shapley algorithm with the magnitude of the cosine similarity score indicating the "preferences" of each vector. These scores are available in Figure 38. If the vectors are stable with a subset of the data, the cosine similarity scores will stay close to one. If they are random, they will have low values near zero, indicating that the vectors do not line up. As a baseline, random data was generated 500 times and analyzed with this technique to calculate a baseline score of 0.1, far below the scores produced by the analyzed dataset. For each of the top 32 components, the distribution clusters tightly, demonstrating that the PCA components are not a localized representation of a subsection of the kernel collection but rather representative of the entire collection.

5.4.4 Summary

Using the preprocessing pipeline demonstrated in Figure 35, it is possible to shrink 920 features generated from programs down into a mere 32 PCA components. This reduction results in no loss in accuracy for supervised learning techniques and does not inhibit unsupervised clustering techniques.

5.5 Kernel Classification

Kernel code should be similar to other kernels of the same class. Compiled code for example considers hardware availability in the ISA. If a CPU does not support floating point instructions, compilation to that platform necessitates transforming it into an alternative computation that is slower. This results in code of a particular type using similar computational patterns that cluster in the input space. This work collects each feature from both the

Table 9: Technique Learning Metrics

Technique	Accuracy	Precision	Recall	F1-Score
Optimal DNN	0.968	0.969	0.968	0.968
KNN-5	0.931	0.937	0.931	0.932
SVM	0.861	0.854	0.861	0.850
Logistic Regression	0.750	0.699	0.750	0.712
KMeans	0.972	0.973	0.972	0.972

static source code and the runtime behavior. The static code informs the required types of computations while the runtime information illustrates what types of computation dominate the runtime of the program. Using these features, it is possible to predict a kernel's label.

The computation can survive an incorrectly labeled kernel. A misprediction would result in a suboptimal selection, leading to either lower efficiencies or a compilation failure. Should a program fail to compile, it is a simple task to select the next best accelerator. If the model selects an inefficient accelerator, the impacts are minimal if it is accurate most of the time.

There is no universal accuracy that is sufficient for predicting a kernel's label. If a developer uses the model as a supplement to intuition, a lower accuracy is sufficient due to manual cross validation. Alternatively, a fully automated system will have no such safeguard so the ML model must be more accurate. Although it is not dangerous to misclassify a kernel, it does have the potential to adversely impact the performance of a program. This has the potential to eliminate any potential gains, making such analysis pointless at best.

All current kernel classifiers rely upon a supervised technique. Hashimoto et. al use SVM[37] and only achieve an accuracy of 65%. Gupta et. al use logistic regression[82] to segment kernels for performance and thus do not have an accuracy measure. Neither of these techniques can predict a kernel's label with greater than 90% accuracy. This work compares four different supervised techniques to predict kernel labels: Support Vector Machines, Logistic Regression, and K-Nearest Neighbors.

5.5.1 Alternative Supervised Classifiers

The current state-of-the-art kernel predictors rely upon logistic regression with posynomial features[82]. When this work applied logistic regression to the dataset, the model only achieved an accuracy of 72.7% with an F1 score of 0.712. Although this model achieves significantly better than chance, it is still incorrect 27.3% of the time.

Support Vector Machines (SVM) is another popular technique for predicting kernel categories. Hashimoto et. al attempted to use SVM. Their model achieved an accuracy of 60%[37]. SVM, when applied to this set of input programs, achieved an accuracy of 85.2% and an F1 score of 0.850. This is a significant improvement from logistic regression and Hashimoto's work, but is still relatively inaccurate.

5.5.2 K-Nearest-Neighbor

K-Nearest-Neighbor (KNN) models excel at identifying classes which are homogenous in their feature space, like kernel labels generated in this work. This work used standard Euclidian distances between each data point from the PCA transformation as the distance metric between samples. Traditionally, in a KNN model, a lower number of neighbors results in overfitting. Figure 39 demonstrates the relationship between the test accuracy and the number of neighbors. Low neighbor counts maintain high test accuracy while higher counts gradually become less accurate. This indicates that the label boundaries require high resolution and are not necessarily continuous. This work selects five neighbors to avoid predicting labels in too small a space while still maintaining high accuracy. Upon prediction, KNN had a training accuracy of 95.6% and a testing accuracy of 93.1%. The resultant model has a final precision 0f 0.937 and a recall of 0.931 resulting in an F1-score of 0.932.

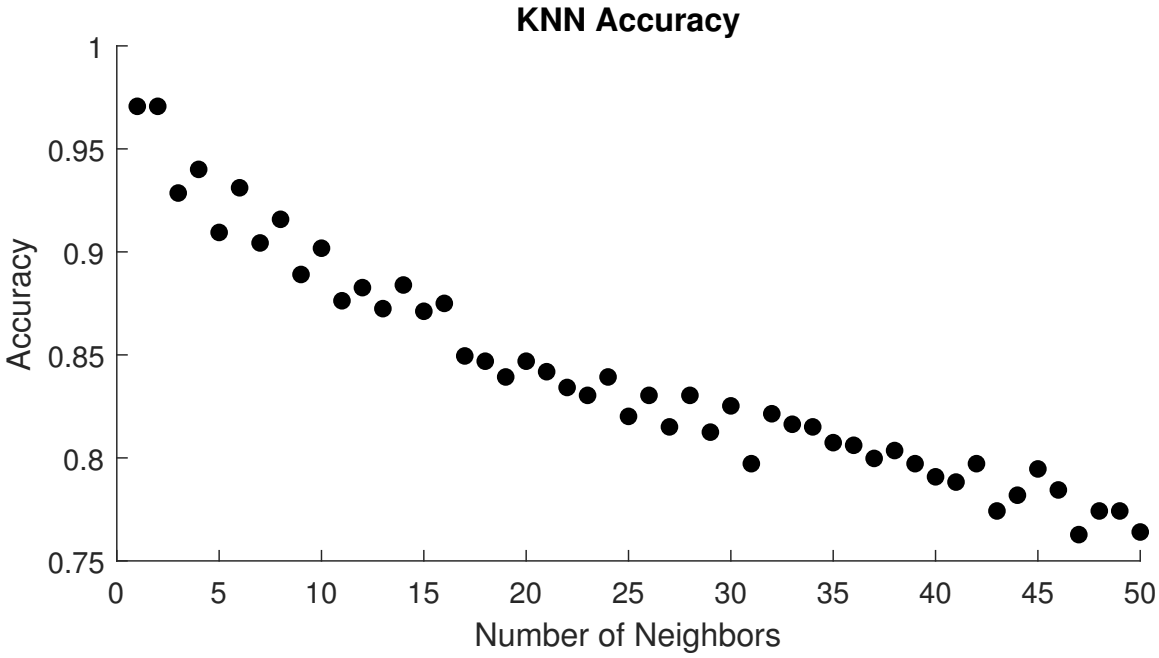


Figure 39: KNN Accuracy
The x-axis refers to the number of neighbors utilized by the KNN algorithm, and the y-axis is the accuracy of the predictor. This work analyzed each KNN with 80/20 cross validation 11 times. The graph reports the median value. As the number of neighbors increase, the accuracy decreases. Without a stability point, the KNN technique is highly fitted to the dataset and unable to predict new data labels.

The KNN model is highly accurate and able to predict all labels; however, the model struggled to predict labels once the number of neighbors increased. Its accuracy drops below 90% after only eight neighbors. This indicates that the label-space does not form coherent clusters and will be difficult to predict with tools that assume a continuous topology. Additionally, despite the cross validation applied, this technique overfits the data (see Section 5.5.4).

Table 10: DNN Design Performance

Wave 1	Wave 2	Wave 3	Accuracy	F1-Score
relu x32			0.958	0.958
Sigma x32			0.957	0.956
relu x32	*Sigma x64*		0.968	0.968
relu x32	relu x64		0.958	0.957
relu x32	tanh x64		0.968	0.968
relu x32	Sigma x32		0.952	0.951
relu x32	Sigma x128		0.963	0.962
relu x32	Sigma x64	Sigma x64	0.959	0.958
relu x32	Sigma x64	tanh x64	0.955	0.954
relu x32	elu x64	Sigma x64	0.968	0.968

5.5.3 DNN Classifier

To develop a DNN model, this work utilizes Keras[86] to rapidly develop, compare, and benchmark various neural network designs. This work also utilizes scikit-learn[87] to create the preprocessing pipeline and evaluate the results. Although this work tested various network operators, simple relu and sigmoid layers achieved the best performance. With the input dataset, this work performed a 50/50 cross validation. Table 10 reports the median of 11 runs. This work waited 5000 epochs for each network to converge.

Within the network, this work varies the number of layers to optimize the network (see Table 10). Adding an additional layer after the input layer did improve performance, but a third layer did not increase the accuracy. This work inserts a dropout layer after every layer with a ratio of 0.2 to minimize overfitting. Similarly, this work finds that utilizing a wave with 64 neurons maximizes the accuracy with additional neurons providing no benefit and fewer neurons hurting accuracy. The network with the highest performance is bolded in Table 10.

The optimal DNN contains three layers (seeFigure 40). The first layer has a rectified-

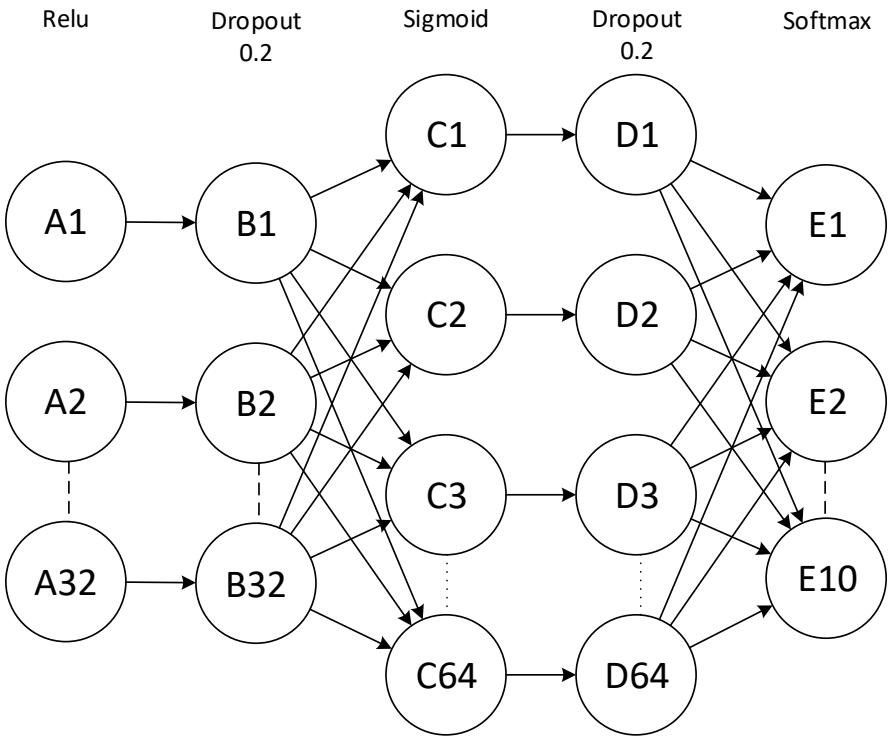


Figure 40: DNN Structure
The optimal DNN structure is a three-layer design with two dropout layers for training. The input layer has 32 relu neurons for the PCA vectors. A dropout layer then discards 20% of the input features during training. Next a sigmoid layer with 64 neurons maps the inputs to an internal kernel representation. Another dropout layer discards 20% of these values before a softmax layer with 10 neurons predicts which of the 10 kernel labels are correct.

linear unit for each of the input PCA features. The second layer of the DNN encompasses 64 sigmoid activators. The final layer is a soft-max layer with one neuron for each output class. The model utilizes two dropout layers with a rate of 20% in training.

To verify that fifty input features are sufficient for robust detection, this work compares the relationship between the feature count and the accuracy of the final DNN (see Figure 41). The first 20 features are vital for the model. Adding subsequent factors in this range results in a significant increase in accuracy. This matches the behavior in Figure 36 where the mutual information score decreases more quickly after this point. The next 30 features

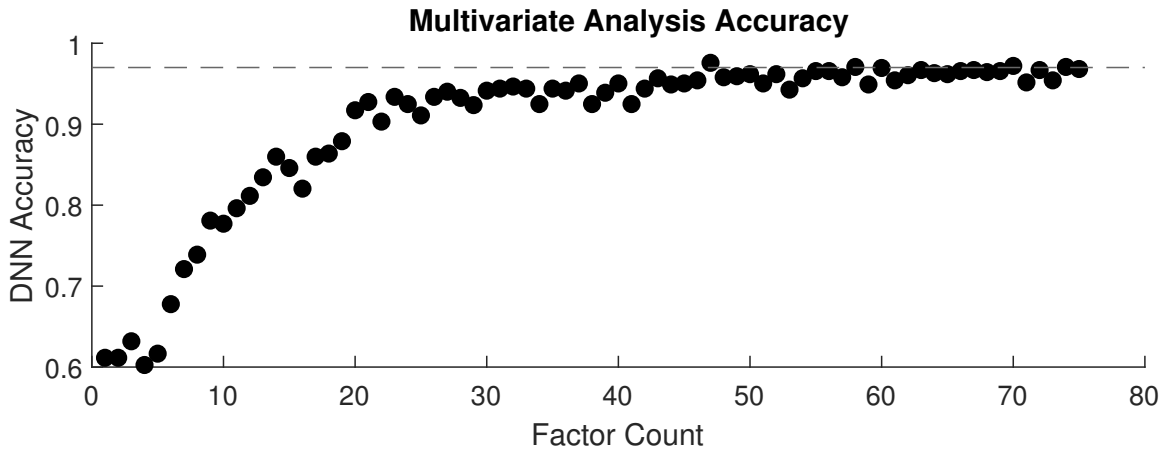


Figure 41: Multivariate Effect on DNN Accuracy
This work performs multivariate feature selection up to 100 factors and used as inputs to the neural network (Figure 40). This work executes each network eleven times. The graph reports the median number. This work gives each training 5000 epochs to converge. The first twenty features are invaluable to the DNN with a significant increase in DNN accuracy for each additional factor. The next thirty features see additional improvement to prediction accuracy, albeit at a slower rate. Features after the top fifty show no additional benefit, indicating the optimal number of features is fifty. The dashed line represents the accuracy level at which the DNN converges, 96.8%.

gradually add to the accuracy of the model before stabilizing at 50 features, matching the break in Figure 36.

Although the model only required 22 PCA components to explain a large ratio of the entropy, the final DNN does not maximize its accuracy with this number of components. Figure 37 shows the relationship between the number of PCA components to the testing accuracy of the DNN. Although 22 components achieve a high accuracy, it does not saturate until 32 PCA components are present with an accuracy of 96.8%. These final components are integral to predict some points, even though these components represent a small proportion of the entropy.

The generated neural network achieves a test accuracy of 95.6% accuracy with a precision and recall of 0.956. This is a significant improvement on the KNN design with an improvement of 1.5%. This work considered various other models at this point by varying

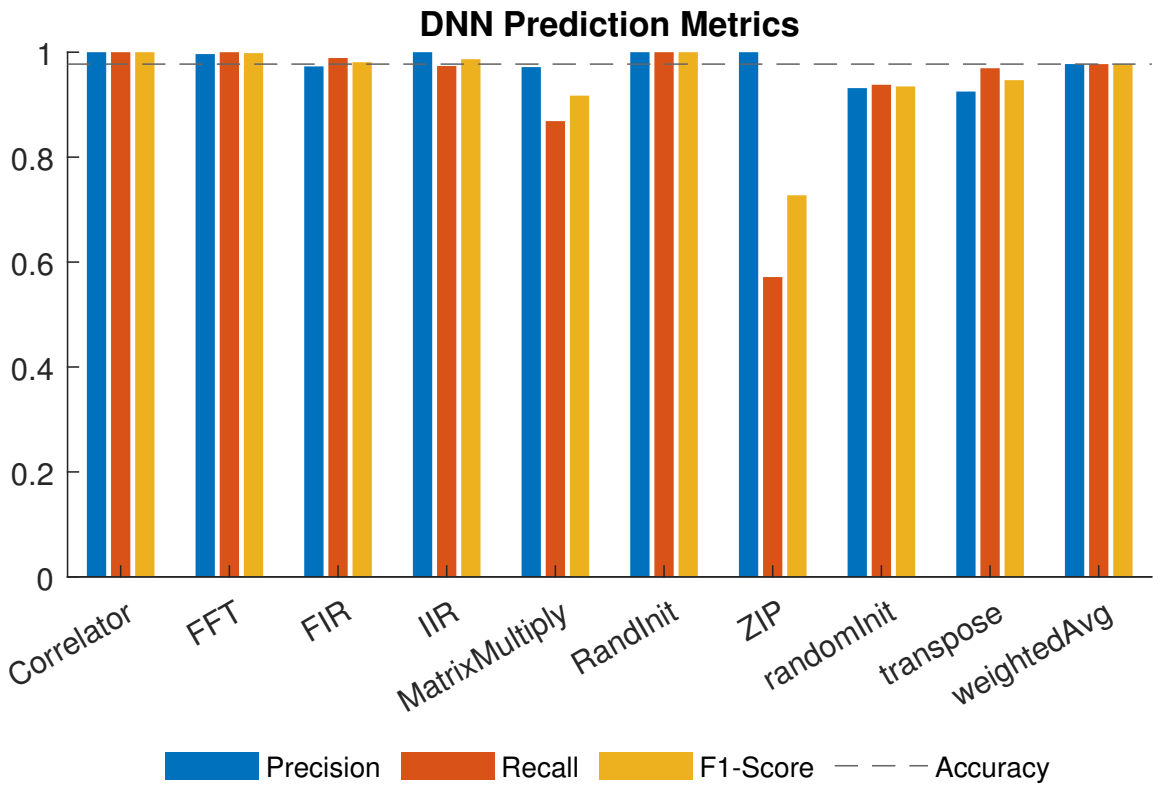


Figure 42: Label Prediction Metrics
The DNN predicted every label but ZIP with high accuracy. ZIP performed poorly due to having a small number of inputs. The model predicted every other label with over 95% accuracy.

the number of layers, activation functions, and number of neurons. Table 10 enumerates a subset of the designs evaluated. Each design present implicitly has a final softmax layer and an alternating dropout layer. The optimal design, in bold, performs better than every other design tested.

Upon analysis, each label has a high accuracy, see Figure 42. This indicates that neural networks are not overfitted for any label. The kernels with the lowest scores are code structures that are not be properly defined as kernels but rather a description of the result of the code.

Table 11: FFT Leave-One-Out Performance

Weighting	Technique	Precision	Recall	F1-Score
Uniform	KNN	0.827	0.731	0.644
Proportional	KNN	0.977	0.488	0.605
Uniform	DNN	0.829	0.923	0.825
Proportional	DNN	0.975	0.985	0.977

5.5.4 Library Analysis

As a further analysis, this work evaluates the performance of the top two classifiers (KNN and DNN) using a leave-one-out cross validation technique that excludes individual libraries. If the results remain positive under these circumstances, the classifier is robust at predicting kernels from code it has not seen before. This more in-depth validation technique is only accurate if a kernel label has many different sources. Due to the nature of the input data, FFT is the only applicable label. The results of this analysis on FFT are available in Table 11. This analysis weighs the scores either by taking the arithmetic average when excluding a library (Uniform), or by scaling the metric by the number of kernels in the excluded library (Proportional). Proportional is more representative of the performance of the predictor, while Uniform is more representative of the performance of any individual library.

The DNN retained its ability to predict FFTs with high fidelity. Model precision and F1-Score decrease by approximately 15% when weighing uniformly. When weighted by population, the performance of the DNN remains excellent with an F1-Score of 0.977. This DNN predicts a kernel's label accurately without sampling similar kernels, something that is important for predicting kernels in wild code.

The performance of the KNN decreases dramatically with this analysis. Although its precision (correct positive predictions) is comparable to the DNN, its recall falls dramatically.

Ultimately, it only achieves a recall (false negative predictions) of 0.731 when weighing uniformly and 0.448 when weighing proportionally. This implies that the KNN began predicting many kernels as FFT to compensate for the lost data. Due to this, its net performance dropped dramatically below the DNN with an F1-Score of 0.644 uniformly and 0.605 proportionally. KNN cannot predict a kernel's label accurately without a sampling of similar kernels, something that will be difficult with wild code.

Most developers train machine learning models against a smaller, limited dataset, leading to overfitting[88]. Generating enough high-quality input data for machine learning is a constant issue. It is common for researchers to perform numerous surveys on kernel-based programs[89][90]. Developers often data mine open-source repositories on websites such as GitHub[91][37] to acquire the necessary data. Simply scraping online repositories provides many low-quality unlabeled programs. Due to the significant effort required in extracting programs, most works utilize a smaller subset of programs to make inferences.

This paper utilizes the same programs in TraceAtlas. The developers continue to add additional programs over time, resulting in a larger data set[30]. By modifying the tracing attribute, it is possible to inject labels directly into the kernel from the source code, which is an efficient, trustworthy way to enter labels[92]. By utilizing this database, thousands of kernels are available for machine learning across a broad swath of the computational landscape. Even though this data set is large, these results demonstrate that the ML model requires more data for the remaining kernel labels to provide robust results that extend beyond the training set.

5.5.5 Summary

Classical supervised prediction techniques struggle to accurately predict a kernel's label from the collection in Table 5. Both SVM and logistic regression fail to surpass 90% accuracy. Neighbor based techniques such as KNN appear to have the potential to accurately predict a kernel's label. Unfortunately, generic KNN appears to quickly overfit to the data (see Table 11), indicating a more robust clustering is necessary. A DNN robustly predicts a kernel's label within the kernel label space. It achieves an accuracy of 96.8% and maintains an F1-Score over 0.8 when predicting FFT from new applications. This demonstrates that it is possible to predict a kernel's label from wild code.

5.6 Conclusion

This work demonstrates that it is possible to predict a kernel's label based on a combination of source code attributes and runtime behavior. This work analyzes over 9000 kernels from 1500 applications. This work quantifies each of these kernels by analyzing both the bitcode and dynamic execution trace. Upon inference, a DNN achieves an accuracy of 96.8%. This demonstrates that it is possible to predict kernels with high accuracy.

The application of multivariate analysis and PCA reduces the 920 input features to a mere 32. Due to the curse of dimensionality, it is ill advised to work with all 920 features directly. A pipeline composed of a variance filter, multivariate analysis, and PCA reduction reduces the dimensionality to 32 with no discernable loss of accuracy.

Lazy classifiers such as KNN can predict a kernels label with high accuracy, but each label does not cluster throughout the space. This limits the technique to only predicting kernels similar to those already labeled. It also indicates that although lazy classifiers can

be accurate, they must be overfitted to achieve high accuracy; thus, lazy classifiers are not appropriate to predict kernels too different from the training set, such as those from wild code.

Predicting the kernel labels robustly requires a more complex classifier, such as a DNN. Other simple classifiers including SVM (86.1%) and logistic regression (75%) cannot properly predict a kernels label with the features gathered in this work. The complexity of the label distribution requires a more complex model such as DNN (96.8%). Ultimately, the model only requires one hidden layer to predict kernel labels accurately. The DNN presented here predicts FFT accurately. The models demonstrated require more data to predict additional labels with high fidelity.

The prediction of kernel labels is the first step in predicting the architecture of best fit. By inferring what a kernel is, it is likely that a predictor could also infer the affinity of a kernel to a particular architecture. With this information, it is viable to automatically compile and schedule naïve code for any domain specific SoC.

Chapter 6

CONCLUSION

Computational kernels are widespread in modern applications. Encryption uses them to iterate over data packets. Image processing uses them to extract useful information from images. Radio processing uses them to transmit messages more efficiently. Linear algebra algorithms use them to segment data. Kernels are ever-present in modern computations. Optimizing these kernels will achieve the greatest return on time investment.

Kernels can be identified and described efficiently through the combination of static binary analysis and dynamic runtime analysis. Most analysis by prior works has prioritized static program analysis. Static program analysis has achieved great gains, but it requires kernels to be preannotated or written in a specific form that has been enumerated in advance. Dynamic program analysis can provide additional information to static analysis, including exact runtime behavior and true memory dependencies. Although traditionally too expensive, this work has demonstrated that it is possible to generate a full dynamic tracing with minimal overhead.

This work has also provides a decidable kernel definition that allows for efficient kernel extraction with only basic runtime information and the source code. The constraints in this definition are written in such a way that they require no additional information beyond the program's CFG and a basic successor matrix. Knowing that all kernels within a program have been identified is combinatorial with this definition, but it can state authoritatively whether a collection of blocks is a kernel. In addition, this definition allows for kernels to be hierarchical, reflecting how programs are written in the real world. With this representation, optimizations are able to work at any level of the stack with minimal changes.

Once the kernels have been detected, it is possible to predict a kernel's type based upon its compile-time and run-time characteristics. This work exhibits how this can be done through the application of a basic DNN. Although the proposed DNN is able to predict labels extremely efficiently, what we think of as kernel labels do not cluster by performance benchmarks. Fundamentally, the way a code is written is far more descriptive of its behavior than the action that a kernel is performing.

Extracting the kernel task graph and identifying those kernels provides an opportunity for speedup through coarse-grained parallelism and tuned kernel code. Across many DSLs, a recurring theme is the ability to structure an application into a representation that allows for the runtime to take advantage of concurrency, data locality, and polyhedral transformation. This advantage translates into orders of magnitude speedups, especially when compared to naive code. For example, providing the correct structure for Bilateral Grid allowed the program to be executed on a GPU, achieving a speedup of 65.2x over the reference implementation[93]. The work proposed in this dissertation facilitates the automatic translation of a reference implementation into the structured DSL representation required for that speedup.

Additionally, simply knowing the identify a kernel allows for trivial code-swaps, which can provide further acceleration. For example, translating a naive FFT to an optimized FFT library[16] or equivalent GPU call[94] could see speedups of 55% and 700% respectively. Similarly, the kernel identity allows the compiler to map application components to fixed-function hardware accelerators, which can be difficult for non-experts to utilize. Simply using an FFT coprocessor[9] would achieve a 3x performance improvement relative to optimal CPU designs.

The primary limitation of this work is that it relies upon runtime information. This implies that a representative sample set of programs must be provided. Should a kernel not

execute in the given trace, it is impossible for these techniques to identify the kernels. This can be compensated for by combining multiple traces, but the technique can still only infer based upon the programs it utilizes.

The algorithms given, although fast, are still not within the strict time bounds required by compilers. Compilers require that the applied algorithm to be high performance to enable frequent execution during development. A slow compiler encourages developers to not use the optimizations, thus defeating their purpose. These algorithms currently cannot be run continuously as a part of the development process, but would instead be used only on the final result. Through its application, additional optimizations can be performed, encouraging application developers to focus their time more on functionality and less on performance.

This work is the first step in enabling the use of advanced program optimization that has been too expensive for most applications to date. These techniques do not require the code be formatted in a particular fashion as many other forms of analysis require. The ability for this to operate on any code allows for other programs, written by naive programmers, to be optimized with high performance optimizations. Knowing what the kernels are and its characteristics allows for a kernel to be compiled more efficiently, scheduled on alternative hardware, and potentially be automatically parallelized regardless of input format.

This work lays the groundwork for future work and discusses the potential for optimizations, but these are left for others to explore. Instead, TraceAtlas is intended to provide the input for these tools. Additional work will be needed to format the kernels in a form these tools can understand and operate on.

The study of the kernels identified will expose a broader understanding of what code is actually being executed, rather than the code we believed was written. Exploration of this could lead to new compiler optimizations, new accelerators, and potentially new computation paradigms. Fundamentally, optimizations currently used in the field are those

proposed by manual inspection. Today, compilers are able to optimize more efficiently than any programmer, as long as the code is written in a way the compiler can understand. Structures that the compiler does not understand are usually kernels that have not been deemed important enough to optimize yet. By extracting them automatically, it will become far more apparent what is actually executing, and therefore important to optimize.

It is possible that kernels proposed here can be extracted within a fixed time bound deterministically. This could allow for kernel annotation to occur as a part of the compilation process. At this point, the algorithm to extract kernels is slow, but the application of relatively common optimizations could easily push its performance to something far more reasonable. By doing this, it will be possible to notify a programmer about the kernels present in their programs, enabling them to write code more intelligently.

Identifying these kernels has the potential to change the way we write, compile, and execute code. Simply identifying kernels will not be sufficient to bring about all of these changes. This is merely a tool that will better inform a programmer of what code is being executed. This is just the first step, with future work being required to fully unlock the full potential.

REFERENCES

[1] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 37–47.

[2] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines." *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[3] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 389–402.

[4] J. Melchert, K. Feng, C. Donovick, R. Daly, C. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, "Automated design space exploration of cgra processing element architectures using frequent subgraph analysis," *arXiv preprint arXiv:2104.14155*, 2021.

[5] "Kernel." [Online]. Available: https://www.merriam-webster.com/dictionary/kernel

[6] O. Segal, N. Nasiri, and M. Margala, "A foray into efficient mapping of algorithms to hardware platforms on heterogeneous systems," *arXiv preprint arXiv:1605.04582*, 2016.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of . . . , Tech. Rep., 2006.

[8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[9] T. Thanh-Hoang, A. Shambayati, C. Deutschbein, H. Hoffmann, and A. A. Chien, "Performance and energy limits of a processor-integrated fft accelerator," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[10] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2010, pp. 365–376.

[11] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, *GNU scientific library*. Network Theory Limited, 2002.

[12] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*, Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013, http://hpc.pnnl.gov/projects/PERFECT/.

[13] P. Karn, "Fec library version 3.0.1," 2007.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735702

[16] M. Frigo and S. Johnson, "Fftw. fast fourier transform library," *URL http://www.fftw.org/*, 2005.

[17] "DAW, Netlib Library," http://www.netlib.org/benchmark/.

[18] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[19] A. Ltd, "mbed tls," 2015. [Online]. Available: https://tls.mbed.org/

[20] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 76–79, 2014.

[21] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org/, 2010.

[22] Audiofilter, "spuce." [Online]. Available: https://github.com/audiofilter/spuce

[23] J. Gaeddert, "Liquid dsp-software-defined radio digital signal processing library," https://liquidsdr.org/.

[24] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016.

[25] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, p. 1–19, Jul. 1970. [Online]. Available: https://doi.org/10.1145/390013.808479

[26] C. A. Lattner, "Llvm: An infrastructure for multi-stage optimization," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2002.

[27] P. B. Schneck, "A survey of compiler optimization techniques," in *Proceedings of the ACM annual conference*, 1973, pp. 106–113.

[28] "Llvm language reference manual¶." [Online]. Available: https://releases.llvm.org/9.0.0/docs/LangRef.html

[29] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[30] R. Uhrie, C. Chakrabarti, and J. Brunhaver, "Automated parallel kernel extraction from dynamic application traces," 2020.

[31] A. Van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of computing and information technology*, vol. 10, no. 1, pp. 1–17, 2002.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.

[33] E. Slaughter, "Regent: A high-productivity programming language for implicit parallelism with logical regions," Ph.D. dissertation, Ph. D. dissertation, Stanford University, 2017.

[34] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron, "Portable programming with rapid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 939–952, April 2019.

[35] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148233

[36] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2584665

[37] M. Hashimoto, M. Terai, T. Maeda, and K. Minami, "An empirical study of computation-intensive loops for identifying and classifying loop kernels: Full research paper," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 361–372. [Online]. Available: https://doi.org/10.1145/3030207.3030217

[38] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[39] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky *et al.*, "Composition and reuse with compiled domain-specific languages," in *European Conference on Object-Oriented Programming*. Springer, 2013, pp. 52–78.

[40] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.

[41] "Nvidia cuda compute unified device architecture." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[42] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[43] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, May 2018.

[44] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun, "Floating-point mixed-radix fft core generation for fpga and comparison with gpu and cpu," in *2011 International Conference on Field-Programmable Technology*. IEEE, 2011, pp. 1–6.

[45] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *ACM SIGGRAPH 2005 Courses*, 2005, pp. 234–es.

[46] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *Computer*, vol. 26, no. 6, pp. 18–27, 1993.

[47] H.-D. Cho, P. D. P. Engineer, K. Chung, and T. Kim, "Benefits of the big. little architecture," *EETimes, Feb*, 2012.

[48] G. Teodoro, T. M. Kurc, T. Pan, L. A. Cooper, J. Kong, P. Widener, and J. H. Saltz, "Accelerating large scale image analyses on parallel, cpu-gpu equipped systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 1093–1104.

[49] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 10 2017, ch. 7. Domain Specific Architectures, pp. 540–617.

[50] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor, "The greendroid mobile application processor: An architecture for silicon's dark future," *IEEE Micro*, vol. 31, no. 2, pp. 86–95, 2011.

[51] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Commun. ACM*, vol. 61, no. 9, p. 5059, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3154484

[52] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "Gramps: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, pp. 4:1–4:11, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1477926.1477930

[53] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[54] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 1045–1058. [Online]. Available: http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/3297858.3304068

[55] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*. New Mexico State University Portland, OR, 2003, pp. 24–27.

[56] M. A. Holliday and C. S. Ellis, "Accuracy of memory reference traces of parallel computations in trace-drive simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 1, pp. 97–109, Jan 1992.

[57] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng, "Efficiently acquiring communication traces for large-scale parallel applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 11, pp. 1862–1870, 2011.

[58] M. Segal and K. Akeley, "The design of the opengl graphics interface," Silicon Graphics Computer Systems, Tech. Rep., 1994.

[59] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[60] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 97–108, Jun. 2014. [Online]. Available: http://doi.acm.org.ezproxy1.lib.asu.edu/10.1145/2678373.2665689

[61] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 535–546.

[62] [Online]. Available: https://ffmpeg.org/

[63] Flang-Compiler, "flang-compiler/f18," Nov 2019. [Online]. Available: https://github.com/flang-compiler/f18

[64] Compiler-Tree-Technologies, "compiler-tree-technologies/fc," Aug 2019. [Online]. Available: https://github.com/compiler-tree-technologies/fc

[65] Microsoft, "microsoft/llvm-mctoll," Jun 2019. [Online]. Available: https://github.com/Microsoft/llvm-mctoll

[66] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Commun. ACM*, vol. 59, no. 2, pp. 114–122, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2863701

[67] S. E. Arda, A. Krishnakumar, A. A. Goksoy, N. Kumbhare, J. Mack, A. L. Sartor, A. Akoglu, R. Marculescu, and U. Y. Ogras, "Ds3: A system-level domain-specific system-on-chip simulation framework," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1248–1262, 2020.

[68] E. Deniz and A. Sen, "Using machine learning techniques to detect parallel patterns of multi-threaded applications," *International Journal of Parallel Programming*, vol. 44, no. 4, pp. 867–900, Aug 2016. [Online]. Available: https://doi.org/10.1007/s10766-015-0396-z

[69] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov 2018.

[70] B. Bollobás and O. Riordan, "Dijkstra's algorithm," *Network*, vol. 69, p. 036114, 1959.

[71] B. Awerbuch and R. G. Gallager, "Distributed bfs algorithms," in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 1985, pp. 250–256.

[72] L. Eeckhout, "Is moore's law slowing down? what's next?" *IEEE Micro*, vol. 37, no. 4, pp. 4–5, 2017.

[73] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[74] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[75] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[76] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[77] J.-Y. Liou, X. Wang, S. Forrest, and C.-J. Wu, "Gevo: Gpu code optimization using evolutionary computation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–28, 2020.

[78] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[79] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, ser. PPPJ '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 27–36. [Online]. Available: https://doi.org/10.1145/2807426.2807429

[80] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 725–737. [Online]. Available: https://doi.org/substitute1

[81] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous cpus," in *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2017, pp. 121–128.

[82] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–20, 2017.

[83] Y. S. Shao and D. Brooks, "Isa-independent workload characterization and its implications for specialized architectures," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 245–255.

[84] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 83–92.

[85] B. C. Ross, "Mutual information between discrete and continuous data sets," *PloS one*, vol. 9, no. 2, p. e87357, 2014.

[86] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[87] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[88] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM computing surveys (CSUR)*, vol. 27, no. 3, pp. 326–327, 1995.

[89] D. E. Knuth, "An empirical study of fortran programs," *Software: Practice and experience*, vol. 1, no. 2, pp. 105–133, 1971.

[90] C. Collberg, G. Myles, and M. Stepp, "An empirical study of java bytecode programs," *Software: Practice and Experience*, vol. 37, no. 6, pp. 581–641, 2007.

[91] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 779–790.

[92] X. Chen, H. Fang, T.-Y. Lin, R. Vedantam, S. Gupta, P. Dollar, and C. L. Zitnick, "Microsoft coco captions: Data collection and evaluation server," 2015.

[93] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Commun. ACM*, vol. 61, no. 1, p. 106–115, Dec. 2017. [Online]. Available: https://doi.org/10.1145/3150211

[94] A. Nukada and S. Matsuoka, "Auto-tuning 3-d fft library for cuda gpus," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–10.