COMPILER-ASSISTED PROGRAM MODELING FOR PERFORMANCE TUNING OF SCIENTIFIC APPLICATIONS

by

KEWEN MENG

A DISSERTATION

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

September 2021

DISSERTATION APPROVAL PAGE

Student: Kewen Meng

Title: Compiler-Assisted Program Modeling for Performance Tuning of Scientific Applications

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

Boyana Norris	Chair
Allen Malony	Core Member
Jee Choi	Core Member
Sara Hodges	Institutional Representative

and

Andy Karduna	Interim Vice Provost for Graduate Studies

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

Degree awarded September 2021

DISSERTATION ABSTRACT

Kewen Meng

Doctor of Philosophy

Department of Computer and Information Science

September 2021

Title: Compiler-Assisted Program Modeling for Performance Tuning of Scientific Applications

Application performance models are important for both software and hardware development. They can be used to understand and improve application performance, to determine what architectural features are important to a particular program component, or to guide the design of new architectures. Creating accurate performance models of most computations typically requires significant expertise, human effort, and computational resources. Moreover, even when performed by experts, it is necessarily limited in scope, accuracy, or both. This research considers a number of novel static program analysis techniques to create performance-related program representations of high-performance computations. These program representations can be used to model performance or to support efficient and accurate matching of computational kernels. We develop two different tools for static analysis-based program representation and demonstrate how they can be used for the optimization of scientific applications.

This dissertation includes previously published and unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Kewen Meng

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Xidian University, Xi'an, Shaanxi, China

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Master of Science, Computer and Information Science, 2015, University of Oregon
Master of Engineering, Computer Technology, 2013, Xidian University

AREAS OF SPECIAL INTEREST:

High-Performance Computing
Compiler Optimization
Machine Learning

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, University of Oregon, 2016-2020
SDE Intern, Facebook, Fall 2019
SDE Intern, Databricks, Summer 2019
Research Intern, Lawrence Livermore National Laboratory (LLNL), Summer 2018
Research Intern, Lawrence Livermore National Laboratory (LLNL), Summer 2017
Graduate Teaching Fellow, University of Oregon, 2013-2016

GRANTS, AWARDS AND HONORS:

Student Travel Grant, Supercomputing (SC), 2017
NSF Student Travel Grant, Cluster, 2017

PUBLICATIONS:

Meng, K. & Norris, B. (2021) From CPU to GPU - A Unified Framework for CPU/GPU Performance Optimization. [In preparation]

Meng, K. & Norris, B. (2020). Guiding Code Optimizations with Deep Learning-Based Code Matching. *The 33rd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*

Meng, K. & Lim, R. & Norris, B. & Malony, A. (2018). Performance Modeling through Hybrid Static/Dynamic Analysis. *Workshop on Modeling and Simulation of Systems and Applications (ModSim).* [Poster]

Meng, K. & Norris, B. (2017). Mira: A framework for static performance analysis. *IEEE International Conference on Cluster Computing (CLUSTER)*, 100, 103-113.

ACKNOWLEDGMENTS

I know that when I start to write this section, this means that my Ph.D. study is about to end. Many times, I tried to imagine what I would say at this moment, but I never thought I would be speechless with emotions. People say that completing a Ph.D. is not an easy task, and I am not an exception. The Ph.D. journey was full of joy, disappointment, excitement, and frustration. Day after day, we kept practicing and training to bravely face the challenges in research and in life.

I owe many thanks to Dr. Boyana Norris. I am so blessed to have her as my Ph.D. advisor. She provided me with tremendous research freedom and always encouraged me to explore various areas and try anything that I am interested in. Boyana patiently taught me how to conduct research, what to do in solving a problem and how to write a paper. Her advising is always on the point, and the editing is elegant. Besides, her positive attitude towards the challenges influences me all the time, especially during the dark time of the pandemic. With her brilliance, integrity, diligence, and caring, Boyana shows what a good scholar and advisor should be. Without her support, I would not have made it this far.

I am fortunate to have Dr. Allen Malony, Dr. Jee Choi, and Dr. Sara Hodges on my committee. Thanks for the significant amount of time they spent and the valuable feedback during my DAC meetings and oral defense. I also would like to thank all the members of HPCL for their generous help. I always enjoy the conversations with Samuel Pollard and Brian Gravelle. I thank Cheri Smith, our graduate program coordinator, for helping us out with all the tedious paperwork and teaching me a lot of interesting things about the culture. I am grateful for having so many good friends here at the University of Oregon.

I take this opportunity to thank the Carnes family. I still clearly remember the day I met with Carl and Karl on the plane, which is the first day I arrived in the U.S. They treated me like one of the family, and we had great times at Thanksgiving and Christmas. It is so hard for me to forget the delicious food they made. Thanks for having me and making Eugene my second hometown.

Thanks to my parents for their unconditional love and support for me to pursue my dream. They encourage me, inspire me, and are being with me to climb over the steep mountain and walk through the stormy sea. Thanks, YW, for the encouragement and being there with me at the beginning of this journey.

Last but not least, I am thankful that my research was supported by the Scientific Discovery through Advanced Computing (SciDAC) program of the U.S. Department of Energy through the RAPIDS Institute.

I dedicate this dissertation to my grandmother.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table Page

CHAPTER I
INTRODUCTION

Performance models are a type of reduced representation of the behavior of programs. They can be used to describe and predict the behavior of an application to provide software developers and researchers with insightful information about the execution status to help them identify the potential bottlenecks to further optimize the performance. Performance modeling techniques fall into three categories by the method of generation and data collection: analytical, empirical, and simulation-based approaches. Analytical modeling involves creating parameterized expressions by abstracting static or dynamic characterizations of an application. Empirical modeling methods rely on performance data obtained during the actual execution of programs on the target architecture. Simulation-based approaches provide a software environment that mimics a computer, on which the program can be "executed". Software performance models obtained by any combination of these approaches are an essential tool for designing and using current and future architectures.

1.1 Motivation

As the development of new architectures accelerates, the computing capability of high-performance computing (HPC) systems continues to increase dramatically. However, many applications cannot fully use the available computing potential, wasting a considerable amount of resources and human effort spent on non-portable optimization. Missed opportunities are partially due to the inability to fully utilize available computing resources or take advantage of specific capabilities of architectures during application development that mainly relies on high-level languages and compiler optimizations. Hence, modeling program behavior at a fine granularity is an essential component of understanding performance bottlenecks and potential of implementations.

Performance models are helpful not only during the implementation of the program but also for the entire software life-cycle. For example, a performance model is able to provide guidance to the application developers in choosing the most suitable algorithm to reach some performance goal. In the phase of application testing, it offers direct feedback of the performance and resource consumption of an application by adjusting parameters. In addition to software development, performance modeling also plays an important role in system design and tuning. Specifically, performance models are capable of describing non-existent hardware, which helps researches to explore the hypotheses when designing new systems. For example, models can help select the best network topology to connect nodes in order to be more efficient for a large-scale parallel system. A performance model can be used to indicate the expected design performance of a platform with a given configuration, which, when compared with the measurement performance by benchmarking, would reveal tuning opportunities. However, performance models are still difficult to create and maintain as software evolves. Hence, automating the generation of performance models would increase their use in practice and can have significant impact on both software and hardware development.

1.2 Research Questions

In this thesis, our goal is to address the following high-level research question through the three sub-questions that focus on specific approach strategies.

What program representations enable the static performance modeling and accurate matching of key computational kernels to support manual or compiler optimizations?

RQ1 Can we use static binary analysis to construct representations of programs for performance modeling?

RQ2 Can we use compiler-generated intermediate representations to create program embeddings that allow accurate matching of loop-based computations?

RQ3 Can we optimize GPU computations based on language and architecture-independent program representation matching?

1.3 Research Approach

In this work, we present novel program representations and performance modeling methods to aid in the optimization of scientific applications. In existing performance modeling tools, a considerable amount of research focuses on dynamic approaches, which rely on runtime information to optimize the target applications, whereas static performance modeling receives significantly less attention. Although performance models based on runtime data are potentially more accurate, descriptive, and easier to generate compared to static methods, their high runtime cost and inability to capture all possible program behavior are significant limitations. In contrast, static performance modeling is preferable when a complete (covering all execution paths) model is needed quickly, or when wishing to model performance on non-existent architectures.

In this thesis, we introduce new program representations and develop methodologies for the generation of performance models and program optimization suggestions through primarily static code analysis. As the input to our analysis, we consider both binaries and compiler-generated intermediate program representations. The novelty of our approach is that in the performance tuning cycle we focus on kernel identification and reuse of existing optimal tuning parameters. Thus the static code analysis in our approach plays a critical role. It extracts metrics and generates reasonable representation to accurately describe the target kernel. Moreover, we leverage machine learning technique on code identification to further reduce the computation costs.

1.4 Dissertation Outline

This remainder of this dissertation is organized as follows.

– Chapter II Background. We categorize the existing modeling approaches into three groups: analytical, empirical, and simulation-based modeling methods based on the performance data collection techniques and the approach for model generation. This chapter provides insightful details about each category and the comparisons among the different approaches. By understanding the development of performance modeling approaches and the advantages and disadvantages of each category, we find the space where we can improve in this area.

– Chapter III Mira. We present Mira [91] framework as our efforts to solve the first research question. In this work, we leverage the static analysis on the combination of the source and the binary code to overcome the problems caused by solely relying on one type of code. The parameterized model achieves the good accuracy compared to the results from dynamic analysis in its estimates of floating-point operations.

– Chapter IV Meliora. We propose the Meliora [92] framework for solving the second research question. Meliora can accurately match loop-based computations based on the graph representation derived from a compiler-generated intermediate representation. By using the matching results, we are able to guide the CPU optimization while greatly reducing the cost of the process for empirically searching the performance tuning space.

– Chapter V Meliora-based Optimization of CUDA Computations. In this chapter we apply the approach described in Chapter IV to improving the efficiency and performance of GPU optimization. We show that the set of program features and control-flow-based graph representation can also capture parallelism-relevant characteristics. We demonstrate significant improvement in autotuning performance for CUDA versions of the SPAPT autotuning benchmark.

– Chapter VI Conclusions and Future Work. We summarize our contributions of this dissertation in the context of our research questions and discuss future research directions.

1.5 Co-Authored Material

This dissertation includes work from previously published co-authored material. This section lists the chapters with the publications and their authors.

– Chapter III is based on the collaboration research [91] between Boyana Norris (UO) and myself.

- Chapter IV is based on the collaboration research [92] between Boyana Norris (UO) and myself.
- Chapter V is based on the unpublished research between Boyana Norris (UO) and myself.

CHAPTER II
BACKGROUND

2.1 Introduction

Performance models are structural representations of program behavior which are descriptive and predictive. Models can. provide software developers with useful information about potential bottlenecks and help them identify optimization opportunities.

Although performance models may emphasize different performance metrics according to specific requirements, they aim to answer several questions in general: What is the estimated execution time of an application? What is the best execution time that an application can achieve? What is the expected execution time if one (or more) application parameter changes, for instance the number of processors? How much memory is required to run the program and achieve the best runtime?

In this chapter, we categorize the performance modeling approaches into three groups, analytical, empirical and simulation-based. We accomplish this by considering the method of collecting performance metrics, the data representation, and the techniques used to process the data in model generation.

2.2 Analytical Modeling Approaches

Analytical modeling seeks to abstract both static and dynamic characterizations of applications by parameterized expressions. It has been widely used in the high-performance computing area to predict system and application performance, provide guidelines for optimization and automatic tuning performance. Importantly, researchers can take advantage of analytical modeling to explore future directions and verify new ideas as it is able to describe theoretical systems or architectures which do not yet exist. Moreover, developers also benefit from applying analytical modeling in each phase of the software development process. Specifically, analytical modeling offers trade-off decisions, such as the CPU computing power, memory bandwidth, and network topology for setting a super-computing system during the requirement analysis, which help with system tuning for target applications in the testing period, and provide valuable feedback for maintenance. In this section, we describe several analytical modeling approaches aimed at improving application performance. In the following paragraphs, we describe manual and automatic analytical modeling, respectively.

Manual analytical modeling is a technique which does not require performance data obtained from the execution of applications. It is built by the analysis of intrinsic characteristics of the target programs, such as the problem size, the algorithm and communication pattern. In addition, the features of the platform which include architectural metrics such as latency, size of cache line, FLOPS rate, and network bandwidth for running the programs are taken into consideration. Then, the analytical model is built by analyzing the relations among all the potential factors which could impact performance.

Analytical models can be constructed through source code analysis. The understanding of the code and experience of the modeler may determine the representativeness and the accuracy of the model.

Clement and Quinn [33] introduced an analytical model for predicting the performance as speedup of parallel scientific applications running on multiple nodes. The goal of this analytical model is to provide performance information for the compiler to assist the program optimization. This modeling method started

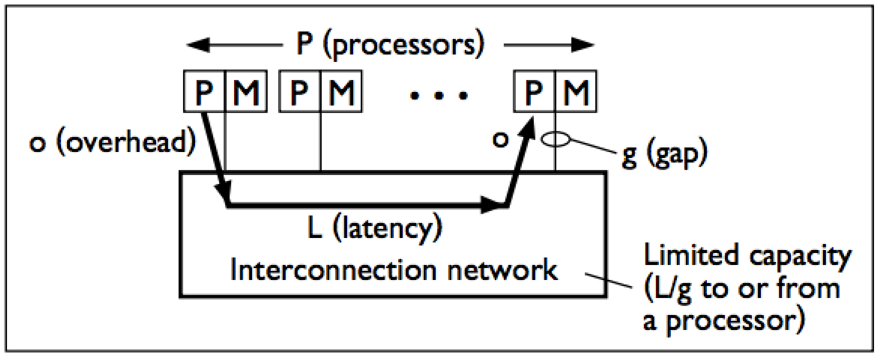


Figure 1. The LogP model. From [34]

from analysis of the previous analytical model [45, 100] for estimating the execution time of parallel applications, which uses the following expression to describe the execution time of an parallel application in general:

$$\tau(p) \;=\; T_s \;+\; \frac{T_p}{p} \;+\; \sigma(p, topology)$$

The term $T_s + \frac{T_p}{p}$ is the ideal execution time of a parallel application. In addition to the ideal execution time, a function of processes and topology is used to represent the communication overhead related to the specific topology of the system. Clement and Quinn focused on startup cost as the major performance impact in the communication overhead, since previous research [57, 39] demonstrated that startup cost is the predominant factor in the overall cost of communication. Therefore, the cost of communication can be expressed as the number of communication times the startup cost: $N_{comm} * C_{startup}$. For instance, the overall communication overhead is: $\sigma(p, topology) = N_{comm} * C_{startup} * (1 + log(p))$ for broadcasting a message on a hypercube topology. Moreover, this model also introduced memory parameter N_m and C_m for representing the number of uncached memory access and the number of necessary cycles for finishing the access operations to the uncached memory. Thus the model is able to estimate the waiting time for uncached memory accessing.

LogP [34] and the extensive work LogGP [2] are two analytical models created for describing an abstract machine configuration revealing the behavior of a shared-memory multiprocessor system. They focus on the characteristics of the interconnection network which is responsible for transmitting point-to-point messages for the communication among all the processors. Figure 1 shows the four parameters used for building the model:

1. $L(latency)$: The upper bound of the communication latency/delay which happens during the message delivery from the source processor to the target processor.

2. $o(overhead)$: The overhead represents the time that a processor spends for sending or receiving the message. During this time, the processor is not able to process any other task.

3. $g(gap)$: The gap indicates the minimum interval in time between two consecutive message sending or receiving for the processor.

4. $P(processor)$: The number of processor/memory modules communicating with each other by messages.

The LogP model is designed for modeling the interconnection communication by the fixed-sized short messages, and it demonstrated its capability of accurately predicting the performance of an algorithm running on the particular architecture. However, with the fast development of the hardware, many architectures (IBM SP2, Paragon, Meiko CS-2) at that time had much higher bandwidth and started supporting long messages in order to further increase the performance. In such scenario, researchers proposed LogGP as the extension of LogP with a linear model specialized for long messages. As a consequence of adding the new component, LogGP introduced a new parameter for the model $G(gap\,per\,byte)$, which quantifies the time each gap consumes per byte for long messages. The reciprocal of G indicates the communication bandwidth for the long message. To validate the LogGP model, authors implemented three long-message scatter algorithms (Simple Long-Message, Binomial Tree, and optimal), and demonstrated good results, comparing prediction with measurement. With appropriate inputs, the logGP model can be used for predicting the performance of parallel algorithms and helping hardware developers explore the impact of the hardware-related parameters on the programs running on a particular architecture. The prediction result can be used to aid the optimization of the existing architecture and design future hardware.

Kerybyson et al. [69] presented an analytical model based method to guide performance and scalability analysis. The goal of this work is to create a predictive analytical model to analyze the performance and scaling behavior of an ASCI [62] application SAGE (SAIC's Adaptive Grid Eulerian Hydrocode). SAGE is a multidimensional (1D, 2D, 3D) hydrodynamics code with adaptive mesh refinement using second order accurate numerical methods. The authors created the model by studying four different aspects of SAGE: The parallel spatial decomposition, the scaling of sub-grid, the common operations in a code cycle, and adaptive mesh refinement. The authors proposed equations to represent each aspect in detail and refined the equations when more factors come into play to ensure that the model is able to reflect the behavior of the application running on the particular architecture. To build the parameterized expressions of the application, two types of data are required. The data indicates the basic metric of the machine, such as latency, MFLOPS rate, and characteristics of the application. The analytical model showed good accuracy for the validation on large-scale architectures: IBM SP3 and SGI Origin 2000, even for future architecture with appropriate machine and application metrics as input.

2.2.1 Manual Analytical Modeling

Manual analytical model parameters described the hardware of a target architecture for running programs and the intrinsic features of applications, which has non-negligible impact on performance. Such parameters are frequently obtained by measurement or from system specification.

In [33], the parameterized model used to estimate application speedup ($S(p) = T_s/T_p$) can be expressed as:

$$T_s = (C_{compile} * (N_s + N_p) + C_m * N_m) * T_{fp}$$

$$\begin{aligned} T_p = {} & C_{compile} * N_s + \frac{(C_{compile} * N_p + C_m * N_m) * T_{fp})}{p} \\ & + \sigma^{'}(p, topology) * T_{fp} + C_{compile} * N_o * T_{fp} \end{aligned}$$

The compiler factor $C_{compiler}$ is the metric for the number of instructions generated after compiler optimization, which indicates the quality of the compiler. $C_{compiler}$ is treated as a constant per compiler, which can be determined by benchmarking the compiler. Similarly, $C_{startup}$ can also be treated as a machine-dependent constant obtained from sampling a program. C_m, which indicates the number of cycles to access an uncached memory location, is obtained from the system specification. The rest of the parameters in the expression are all static parameters which account for the features of the application. The values of those parameters are available for the compiler by parsing the abstract syntax tree of the target program. Researchers can utilize this modeling method during compile-time for prediction-based optimization.

Kerybyson et al. [69] derived the mathematical model by understanding the four components mentioned previously and their underlying relations. Therefore, the performance-related parameters are derived from code analysis about the algorithm, the data structure, communication pattern and operations. On the other hand, the architectural hardware metrics of the machine used in the model, such as the number of the CPU cores can be obtained from the manufacture manual. The complete performance model for the SAGE code can be created by combination of the parameters from software and hardware.

In LogP [34] and LogGP [2], the authors studied the major components in distributed processing, the characteristics of messages transmission, and the nature of interconnection network as the carrier for delivering messages among processors in order to understand the roles of each part and their correlations. LogP and LogGP models simplified the message passing procedures into the parameterized expressions with only four or five machine-dependent parameters. As the author mentioned, the parameters are not equally important in all situations. Therefore, it is possible to ignore one or more parameters depending on the situation. For instance, when the overhead dominates the gap in particular machines, it is reasonable to eliminate the parameter g to simplify the model. All of the parameters used in the model can be obtained by measurement. Thus, it must re-measure the parameters in order to apply the model to evaluate different architectures.

In Tudor and Teo's paper [131, 132], they discussed their work on leveraging analytical modeling technique for analyzing the speedup of shared-memory programs running on multi-core architecture. To understand the behavior of shared-memory, authors divided the entire life span of a running program into *useful* work and overhead, and mainly focused on modeling issues that caused the speedup loss. The speedup model abstracts the performance loss into two parts: speedup loss due to data dependency and memory contention. Data dependency represents the number of inactive threads due to synchronizations, imbalanced workload, or lack of work to utilize all threads. To evaluate the data dependency model, the parameters are inferred by a baseline run in which it executes the program with larger number of threads than processors cores. The outnumbered threads would cause the extra threads to queue in the run-queue. By sampling the number of active threads (executing ones and those in the run-queue) in a constant time interval and the service time of the threads, the time weighted average is obtained. The memory contention model focuses on the growth of stall cycles due to memory contention considering UMA (unified memory access) and NUMA (non-unified memory access) policies respectively. The key parameter in the memory contention model, $C(n)$ the number of cycles, is obtained by measurement. The limits in the memory model is that it only predicts the contention for the memory nodes local to active CPU nodes.

2.2.2 Automatic Analytical Modeling

Manual analytical modeling requires analysis of the application and the interaction with hardware, resulting in a heavy burden of time and human efforts.

```
!$pal def n_px = ${npe_i}
!$pal def n_pointx = ${n_pointx_gbl} / n_px
!$pal def n_py, n_pointy...
! def sweep3d_mdl(g, p, s) = n_solve * (sweep3d(g, p) + s)
! def sweep3d(g, p) = p_fill(g, p) + n_octant * p_octant(g, p) + p_drain(g, p)
! def p_octant(g, p) = (n_sweep − 1) * p_stage(g, p)
! def p_stage(g, p) = (g + (4 * p))
!$pal def p_fill(g, p) = (n_px + n_py − 1) * (g + 2 * p)
!$pal def p_drain(g, p) = (2 * n_px + 4 * n_py − 2) * (g + 2 * p)
!$pal loop n_solve = ... ! hide actual n_solve loop
!$pal model sweep = p_fill(@{grind}, @{post1}) + @{sweep} + \
                    p_drain(@{grind}, @{post1})
      call sweep(...) ! sweep() is the right column
!$pal endmodel
!$pal endloop
```

Figure 2. Program annotation used in Palm framework [126].

To alleviate the burden, several researchers made efforts to automate the modeling procedure through static program analysis.

Narayanan et al. proposed PBound [102], a framework for automatically estimating best case performance bounds of C/C++ applications through static compiler analysis. PBound collects information and generates parameterized expressions for memory and floating-point operations combined with user-provided architectural information to compute machine-specific performance estimates. PBound solely relies on source code analysis, and ignores the effects of compiler transformation, frequently resulting in bound estimates that are not realistically achievable.

Kerncraft, created by Hammer et al., is a static performance modeling tool with the concentration on memory hierarchy. Kerncraft characterizes performance and scaling loop behavior based on Roofline [139] or Execution-Cache-Memory (ECM) [61] model. It uses Intel Architecture Code Analyzer (IACA) [63] to operate on binaries in order to gather loop-relevant information. However, the reliance

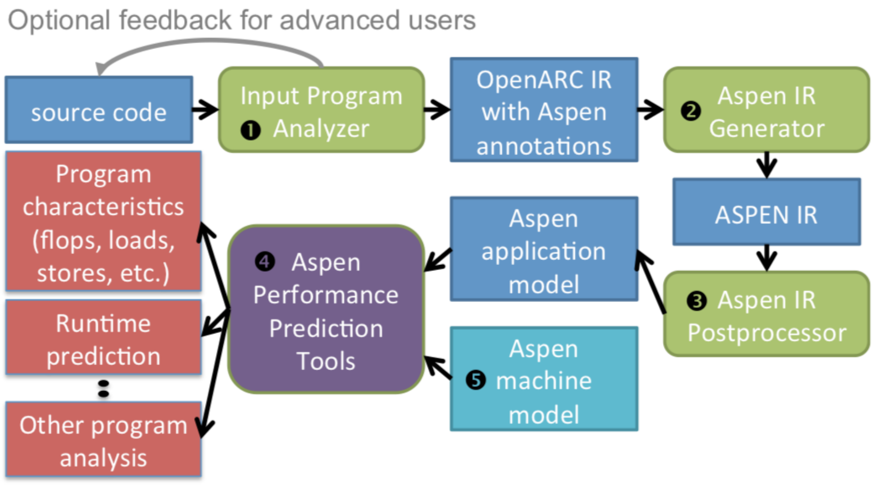


Figure 3. Workflow of the COMPASS framework. From [79]

on IACA limits the applicability of the tool. As a result, the binary analysis is restricted by Intel architecture and compiler.

ExaSAT [30, 133] automates the procedure of performance model generation by employing static code analysis. ExaSAT is built on top of ROSE compiler framework [112] to generate AST (abstract syntax tree) which is consumed by the compiler analysis component to produce loop attributes, and analyze data access. The compiler analysis outputs an XML format intermediate representation which enables the framework to model the hypothetical code by the given XML-IR. However, ExaSAT uses an ideal model to simplify the memory hierarchy and the analysis only works for specific source code pattern (loop type).

To reduce the difficulty of analytical performance modeling, Palm [126] leverages static code analysis to build performance models in Ruby automatically. The resulting model can be evaluated by providing various values. However, it is not a fully automatic framework since it requires users to annotate in order to describe the application (Figure 2). The source code annotation can be a time-consuming task and demands the users to understand the code. Moreover, the model parameters are obtained by measurement from running the instrumented code on specific architectures. Figure 3 presents the COMPASS [79] framework, which shares the similar design ideas of performing the static code analysis to generate performance model. OpenARC [80] is used to examine the source code, including interpretation of the OpenMP and OpenACC directives to identify parallelism. The Aspen IR processor accepts the output of OpenARC to generate the Aspen performance model [122]. Comparing to Palm, COMPASS achieves better automation by using light-weight annotation and the static data enables it to be able to apply to considerably arbitrary code.

2.2.3 Advantages and Disadvantages

Analytical modeling techniques do not rely on performance data from execution to generate models. It summarizes the features of the applications, such as data structure, data distribution, data dependence, and considers static performance-related metrics of hardware such as the cost for message passing, memory access, initialization overhead, CPU cycle. In addition, it abstracts the behavior of the software and the interaction between the software and hardware. The software/hardware metrics are combined, which generates parameterized expressions for performance prediction or diagnosis.

Since analytical modeling requires no performance data from the multiple runs of the target program, it avoids the time-consuming work of designing the inputs, configuring the experiment environment and waiting for performance data, which could take a long time. The properties of analytical modeling enables integration in a compiler as a plugin for guiding optimization. If an analytical model consists of the parameters which compiler is able to obtain statically, then the parameters can be used by the compiler during the compilation of a program to predict the performance and provide feedback to compiler so it can decide the optimization type and targets while maintaining a reasonable compilation time. In addition to the time efficiency, analytical modeling is applicable in every phase of software/hardware development because it is data-independent. Analytical models can help make decisions in design, implementation, and testing. Moreover, comparing with other modeling methods, analytical modeling is more resilient to the changes of the application. Researchers can adjust the parameters and the logical relations regarding the changes in a timely manner, which improves the efficiency of performance evaluation and optimization significantly.

However, to model an application and its underlying communication behavior with the lower-level hardware, one needs expertise in order to obtain a deeper understanding about algorithms used in the application, as well as the technical details of the target architecture. For example, in order to model an application using message passing for communicating among the distributed-memory, it is required to understand the mechanism of message passing, the involved hardware, data organization and partition at a higher-level. The user needs to identify which factors may affect the performance and in what manner, so that they can be correctly placed in the parameterized expression. Furthermore, building the analytical model also involves making trade-off between the model accuracy and the number of model parameters. A great number of parameters indicate that the model considers more factors that will impact the performance, leading to a more accurate result. On the other hand, a model containing fewer parameters at a higher abstract level provides less comprehensive results yet is easier to build. Therefore, the analytical model should be designed carefully to make the balance between complexity of model construction and accuracy of the results. In general, the simple analytical model is suitable for the fast study of an algorithm while a more sophisticated model satisfies the need of deep optimization and tuning of an application.

Although static code analysis based methods automate the analytical modeling to some extent, it may still require human efforts for manually processing the worst case entire source code. This constraint worsens the cost of modeling for complex and large-scale applications.

2.3 Empirical Modeling Approaches

Empirical modeling methods rely on the performance data from actual execution of programs on the target architecture. Therefore, the procedure of empirical modeling can be divided into two phases. First, the program is executed on the target platform to generate performance metrics. The data for characterizing the dynamic behavior of the program is then processed. In this section, we will discuss the empirical modeling approaches according to the two phases: data collection and model generation.

2.3.1 Performance Data Collection

To study the performance of an application, the first step in empirical methods is to run the target program on a particular platform and obtain the data so that we can learn what is happening at runtime. Performance data collection is defined as the methods used to extract necessary performance information. If we consider an execution of the program as a series of various types of events occurring at certain point of time, the goal is to obtain the insight about such events, such as time, duration, and interactions. In the following section, we describe two major techniques for data collection.

1) Code instrumentation. The target executable used for generating data is expected to provide more information for each run given the fact that it could be costly, especially on supercomputers. Most importantly, the unmodified program is only able to provide coarse-grained information such as the execution time or results. Extracting lower-level data about the underlying hardware, such as instruction count, to study the performance is necessary to understand application behavior. Code instrumentation provides an opportunity to probe the application during runtime. Instrumentation modifies the source code by inserting annotations at interesting spots such as the entry/exit point of a function, inside a loop, or around branch statement. When the modified program is executed at the annotation points, the additional function calls are triggered to record the events and related metrics, such as how many time this block has been visited and how long a loop takes to complete.

However, considering the scenario that source code may not be accessible, one can perform runtime instrumentation on the binary executable to modify the binary using dynamic instrumentation tools [121, 85, 20, 103] or performance analysis tools [117, 1] which includes a binary component. Importantly binary instrumentation separates the target program from the platform, offering an opportunity for understanding the code in an architecture-independent manner. Information in the DWARF section helps map binary code back to the source so that users could locate the blocks or the particular functions. However, the compiler optimization would complicate binary instrumentation and makes the implementation of that more challenging.

In addition, there are several tools [101, 134, 46] provided runtime library-level instrumentation other than application-level. Such tools are implemented as libraries between the application and the runtime library. They employ the standard profiling interfaces (PMPI, OMPI) to intercept and record the events and forwards them to the runtime library. Applications are required only to re-link with the instrumentation library.

2) Performance hardware counter sampling. Code instrumentation is able to provide detailed dynamic information about an application. Yet code instrumentation is intrusive and time-consuming. Sampling tools, for example [19], offer an alternative for collecting the performance data without modification of the code. They rely on hardware performance counters which are a part of modern processor performance monitoring units. The performance counters work as a set of registers to record the occurrences of specific signals. When a given interval is reached or the counters overflow, the sampling tool interrupts the running application in order to collect the addresses of the events. It is able to provide information at a fine level of detail, such as operation count, cache misses, pipeline stall. Sampling provides the users with flexibility to make trade-offs between overhead and resolution of the events. By reducing the length of the observation interval, more details about an event can be inferred yet with higher overhead. On the contrary, sampling at a low rate by increasing the number of observation intervals may lead to overlook infrequent events.

Before applying the performance analysis tool, the collected instrumented and sampled data need to be processed to generate either a profile or event trace [96] for data analysis. In a profile, measured events are aggregated together which represents a mapping from events to source code at the statement, loop,

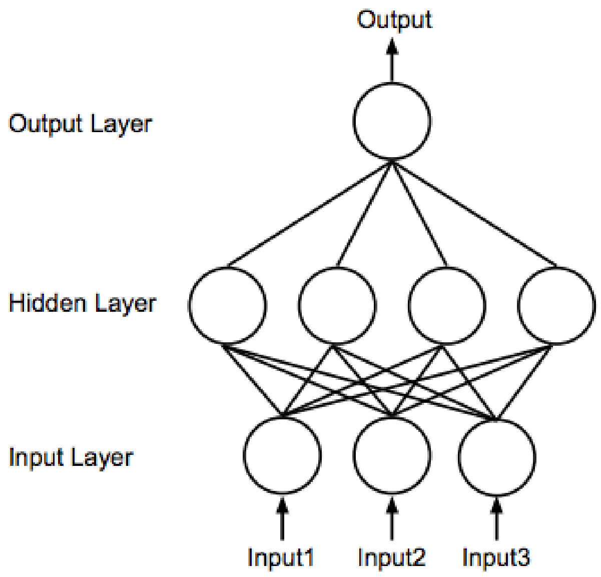


Figure 4. A fully connected feed-forward ANN. From [65]

or function-level. A callpath profile [52], which contains information about the costs and function calling context, is frequently used in performance analysis tools [117, 1, 71]. Traces comprise time-stamped events, which is able to provide more detailed information about the application behavior and suitable for examination of timing-related issues in the program. A profile can be reconstructed from a trace, however constructing a trace from a profile is difficult since sampling may miss events occurring outside of samples. However, the size of tracing data increases with the runtime. Thus, tracing-based analysis may take a large amount of storage.

2.3.2 Model Generation

Once performance data is obtained, the correlations between the input parameters need to be understood. In the follow paragraphs we will cover the approaches for building models.

1) Artificial neural networks. An artificial neural network (ANN) is a machine-learning technique for discovering the correlations between input parameters and results. An ANN comprises a set of nodes and weighted edges connecting the nodes. A node representing a neuron is called a unit in ANN that communicates with other connected nodes by receiving and passing values. There are three types of units in an ANN architecture located in distinct layers along the information flow. Input units are responsible for accepting the input and passing the weights to according units pointed by the outgoing edges. The output unit receives the value and presents the prediction results. Among all the units, the hidden units are of great importance as they carry all the computation tasks, which receive input from incoming edges to process the new output and then passes to either hidden units or output unit. Specifically, after a new training sample is accepted and distributed by the input layer, every hidden unit takes the sum of all values from the previous layer and the weights on the incoming edges, and computes the new sum by the activation function as its output is passed to next layer. Figure 4 demonstrates a fully connected feed-forward neutral network with only one hidden layer in which each unit is connected by every unit in the previous layer. The data flows from the bottom to the top.

In [65] authors employ fully connected feed-forward neural network to build the model in order to predict the program performance. In the feed-forward ANN (as shown in figure 4) the *sigmoid* function is used as the activation function to

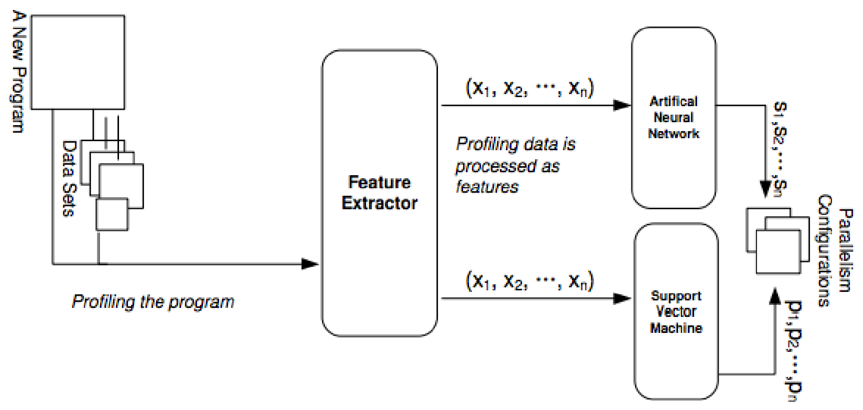


Figure 5. The workflow for parallelism mapping. From [137]

calculate the new output on each hidden unit. By monopolistic executions to reduce the noise in the generate performance data and the applying the appropriate sampling technique and learning mechanism to minimize the percentage error during training, the neural network achieved 5%-7% error on performance prediction of SMG2000 [43].

Li et al. [81] proposed ASpR framework for predicting the performance of runtime parallelization and task scheduling. The Stuttgart Neural Network Simulator [144] is used to build the neural network and make predictions. As an on-line system, ASpR continuously collects runtime information to generate training data including function name, parameter sizes, and measured execution time. In the implementation, authors used a two-hidden-unit (single hidden layer) neural network with sigmoid function as the activation function and update the edge weights by back-propagation algorithm. In addition, to adapt the continuous new available data points, it limits the training data in a sliding window range with reasonable size. And ASpR incrementally retrains the ANN with data from the growing sliding window to improve the prediction accuracy.

Figure 5 presents Wang and O'Boyle's research [137] in which they adopt ANN to build model for predicting the performance of a mapping of program parallelism to multi-core processors to guide the selection of the best thread number for a parallel program. In the proposed approach, the authors profile the instrumented serial version of the program and compiler-parallelized version respectively to obtain code features and runtime feature. Code features covers the characteristics of the target program including cycles per instruction, the number of branches, load and store operations, while the runtime feature mainly refers to the parallel execution time. An ANN with 2 hidden layers (each has 3 hidden units) using Bayesian regularization back-propagation as the training algorithm is constructed to predict the speedup. Moreover, authors use a mutli-class support vector machine (SVM) to predict the best scheduling policy with different number of threads. The combined results can reveal the correlation between program scalability and the number of threads.

2) Regression. In [67], Joseph et al. proposed a method for processor performance analysis based on the linear regression model. The goal of the research is to model the relationship between processor performance and architectural parameters. In the development of their model, they assumed no prior knowledge and understanding about the architecture and the workflow of the processor. Their modeling technique quantifies the significance of architecture-related parameters and the interaction among them. To identify the performance impact of the parameters, they adjusted the value of a parameter, then monitored the performance changes in the simulation results. For example, the measured average instruction issued per cycle can demonstrate the changes in out-of-order issue would cause more performance loss than an in-order issue in terms of performance effect

on the L1 data cache. The authors used 26 key micro-architectural parameters, such as pipeline depth, issue queue size, L2 cache size and issue order, to construct the model. To obtain the best model, the authors performed an iterative procedure to refine the initial model, which starts from a small set of execution results, and then apply the model to guide further data until it reaches the designed error bounds. The model is constructed by following Akaike's Information Criteria [116] to fit the data while maintaining a minimum number of parameters. However Harrell [54] stated in his research that such stepwise regression has several significant biases, which motivates researchers to apply other regression methods statistical modeling.

Lee et al. [78] presented several methods for exploring the parameter domain of larger applications and building predictive models. In this research, the authors demonstrated the effectiveness of applying statistical methods for studying the relationship between the parameters and the performance impact. Hierarchical clustering provides information about the correlations of the parameters. Clustering also filters the redundant predictors for regression model by selecting the most representative predictor that could reflect the cluster's impact on the result if multiple predictors are correlated and clustered into the same group. Association analysis and correlation analysis are used for investigating and quantifying the association between the predictors and the response. It provides the researchers a high-level perspective to understand the parameter domain. The authors compared the prediction between a piece-wise polynomial regression model and the artificial neutral network, and offered suggestions for the selection of the two method. The performance data for building the model is obtained by sampling the performance measurement. To reduce the high costs of exhaustive measurement, authors applied several sampling technique to data collecting. Sampling Uniform at Random (UAR) approach has a full range of parameter values and could ensure the sampled data unbiased and representative for the parameter domain. Stratification minimizes the divergence in relative error for the small performance value with large relative but small absolute errors. It mitigates the bias of the sample data caused by the optimization to reduce the sum of square errors in regression and neutral networks. Regional sampling combined with regression generates per-query regression model according to the similarity (Euclidean distance) of the points to the query.

In Lee's other researches [76, 77], they utilized the same data analysis techniques for the exploration of the correlations among parameters in a larger design space of one billion points for performing the polynomial regression modeling accurately and efficiently for predicting the performance and power. Turandot [98], a parametric, out-of-order, super-scalar processor simulation framework is used to obtain power estimates. Turandot is able to accurately estimates the power consumption by the circuit-level power analysis and resource utilization statistics [18]. Similarly, they applied sampling uniformly at random to collect the number of data points needed for constructing the model in order to reduce the expensive experiment runs.

Barnes et al. [7] described a regression-based modeling method for performance prediction. The modeling technique [7] predicts the execution time of an application on p processors by the execution time collected for the same application running on the architecture with a smaller number (q) of processors, where $q \in \{2, ...p_0\}$ and $p_0 < p$. Instrumentation is used for collecting the execution time of an application. Specifically, it varies the configuration of the inputs and then applies them to the instrumented code to obtain the data for multiple runs. With the performance data, the method then performs a regression to fit the data into a linear prediction model. Three techniques are demonstrated in their work for prediction. The most straightforward one is to simply fit the data from execution by regression and then extrapolate to a larger problem size. It results in a reasonable prediction with a second-order polynomial function for fitting. The second technique separates the regressions of computation and communication by selecting the most representative pair from the per-processor data. The third method constrains the communication time to include no blocking using the global critical path.

Both neural network and regression can provide reasonable accuracy. Regression-based methods offer a statistical perspective on the parameter space. In the above research, authors statistically analyze parameters to explore the correlation among them and such analyses also help to identify the significance of the parameters that contributing majorly to the performance. On the contrary, the neural network presented in previous paragraph did not demonstrate the feature selection of the input data, which may result redundancy in the generated model. However, neural network based approaches achieve better automation than the regression based ones. Users could choose depending on their requirement.

2.3.3 Advantages and Disadvantages

Empirical modeling techniques rely on the performance data from running of the program on the target hardware. One significant improvement for empirical modeling, compared with those do not require actual data, is accuracy. The continuous introduction of new technologies into hardware manufacture and software design enhance the computing capability and efficiency, yet also make it difficult for one to understand the whole system. In this scenario, without vast expertise and rich experience one may fail to consider every detail in code-analysis-based modeling procedure which consequently results in the inaccurate output. Performance metrics collected from actual runs undoubtedly are the most representative data for demonstrating the runtime behavior of a program. Such data enables empirical modeling approaches to reflect the characteristics when running on the real hardware. Profiling by either instrumentation or sampling offers users the different granularity of the program representation to make trade-offs between overhead and accuracy. Furthermore, when the source code is not available, alternatively binary can be instrumented to generate data, which increases applicability of the modeling. The reduced requirement for the expertise on the program enables people with other backgrounds to be involved in the process of model building. For instance, statisticians and machine-learning professionals could focus on the how to construct models based on the given data. Moreover, with the appropriate design it is able to automate the whole procedure of modeling to reduce the laborious work and increase the efficiency.

However, actual execution on the real architecture also has limitations. First, the target hardware have to be available at the time of experiment, which could be a problem for those who cannot afford such machines (e.g large-scale parallel system) or have no access to the particular architecture. Moreover, it is possible that such machine does not exist if the modeling serves as the preliminary study of performance for a program on the hypothetical machine. In addition to hardware accessibility, model portability should also be considered in choosing the modeling approach. Since the empirical methods construct model is based on the application-specific and architecture-specific data, the model may not apply to other programs or hardware which demands duplicated work for re-modeling. Even if the target hardware were available, execution of the program is time consuming, especially for scientific applications that are computation-intensive.

2.4 Simulation-based Modeling Approaches

Performance modeling could be extremely difficult in a real-world scenario for the parallel applications running on the large-scale supercomputers. Due to the large problem size, many factors and the effects among them, such as the implementation of the applications, compiler optimization, interconnect communication pattern, have to be considered for accurate modeling. Moreover, applications may exhibit completely different behaviors in run-time from what was assumed theoretically. Such reasons complicate the performance modeling of parallel application.

2.4.1 Data Representation for Simulation

In this section, we mainly discuss the types of data collected and used in simulation in terms of modeling and analysis. The design of the simulator decides the type of data to collect and process.

1) Application trace from execution. Application tracing is a performance information acquisition approach widely used in HPC domain in order to characterize the application. The application traces is comprised of a sequence of events of computation and communication operations captured during the execution on the real hardware. The events are labeled by time so that the

simulator is able to "replay" the execution to perform the analysis afterwards with various configurations.

PSINS [129] is a typical example of trace-driven simulator for performance modeling and prediction of MPI applications. PSINS consists of two major components: tracer and simulator. The tracer collects detailed information about MPI calls based on MPI's profiling interface (PMPI). In addition to MPI function calls, it also traces the communication and computation time for events. To reduce the resource consumption, PSINS generates a separate event tracing file for each MPI task and only dumps the tracing information when the buffer located for the task is full. Then the separate tracing file will be combined into a compact tracing file after the execution phase is done. After a tracing file is generated, the simulator takes the event tracing file as input along with a set of parameters for modeling the target architecture. The simulator considers the entire architecture as computation nodes connected by global buses. According to the simulation setup, the required parameters come from two different perspectives. The system level parameters include the number of computation nodes, the bandwidth and latency of the bus for the communication between two nodes. The node level parameters are the number of processing units, the number of incoming/outgoing links connected to the buses, bandwidth and latency for the node and the CPU ratio for describing the computation workload.

Carrington et al. [28, 27] proposed a framework for predicting the performance of scientific applications and verified its capability on LINPACK [38], POP (Parallel Ocean Program) [119], NLOM (Navy Layered Ocean Model) [136]. The authors designed the model based on the hypothesis that a single processor performance and the inner-connected network are the predominate factors affecting

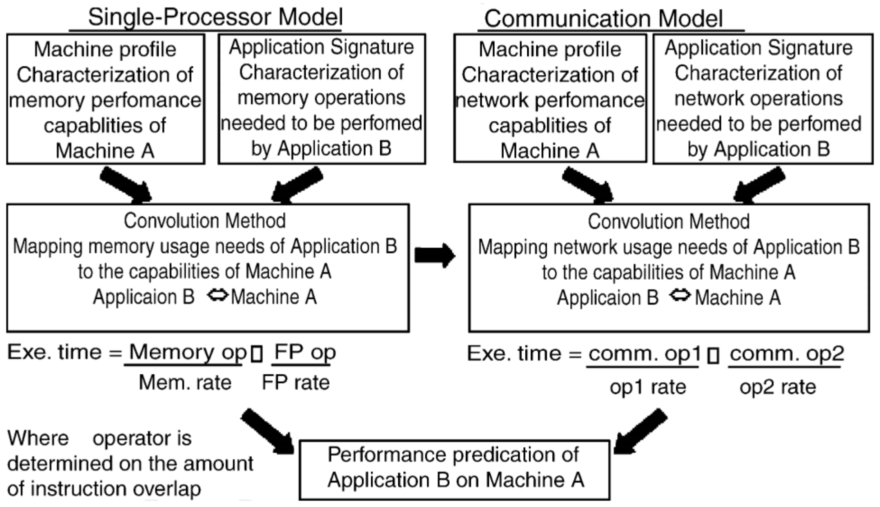

Single-Processor Model
Communication Model
Machine profile Characterization of memory perfomance capablities of Machine A
Application Signature Characterization of memory operations needed to be perfomed by Application B
Machine profile Characterization of network perfomance capablities of Machine A
Application Signature Characterization of network operations needed to be perfomed by Application B
Convolution Method Mapping memory usage needs of Application B to the capabilities of Machine A Applicaion B ⇔ Machine A
Convolution Method Mapping network usage needs of Application B to the capabilities of Machine A Applicaion B ⇔ Machine A
Exe. time = Memory op / Mem. rate ▯ FP op / FP rate
Exe. time = comm. op1 / op1 rate ▯ comm. op2 / op2 rate
Where operator is determined on the amount of instruction overlap
Performance predication of Application B on Machine A

Figure 6. The Performance modeling and prediction framework. From [27]

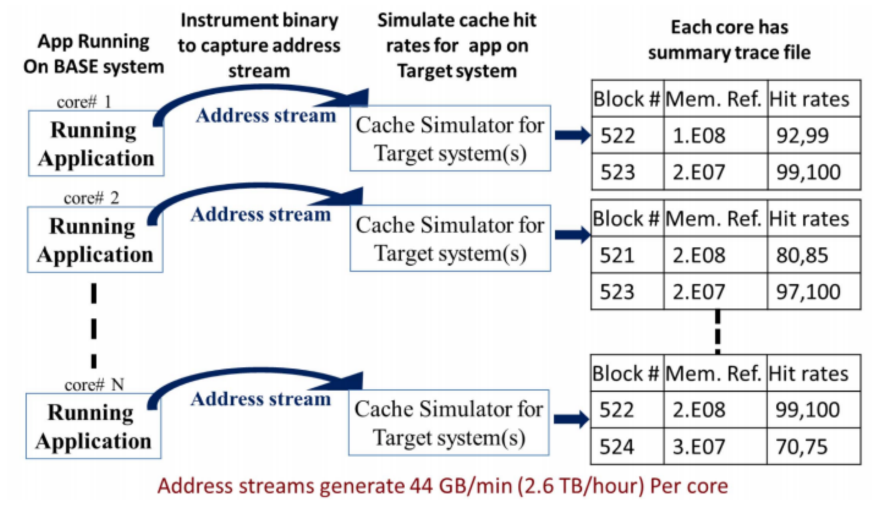
the overall performance of the system. The two factors are sufficient to build the performance model to make a prediction with a reasonable error rate (∼10%) [120]. Therefore, the authors divided the modeling problem into two sub-models representing the two major factors, single-processor model and communication model (Figure 6). Both models are built on the application signatures and machine profiles. Application signatures generated from tracing characterizes the application performance as the potential workload on the processor and the network, such as the memory and network operations. A measurement-based machine profile reflects the ability of the hardware, which includes the rates of message passing, memory load/store operations, and FLOPS.

With extrapolation, researchers utilize an application trace to model and predict [26, 59] the the application performance on large-scale systems. Hoefler et al. [59] presented the LogGOPSim, a simulation framework, for studying the scalibility of the large-scale MPI algorithm. The design of of this framework is to simulate large-scale applications with more than 8 million cores. It focuses on two major parts: the communication and the LogGOPS model. The framework uses a dialect of the Group Operation Assembly Language (GOAL) [60] to efficiently express the collective communication in parallel applications, and a tuple with message tag and source combined with order to match the messages to simulate the message passing semantic. LogGOPS model extends the LogGPS model with minor modification as the authors argued that the LogGPS model only consider a constant overhead per message send, which is not applicable for some architectures. LogGOPS model introduces a new parameter O to model the message transmission overhead per byte. In addition, LogGOPSim provides an extrapolation scheme to study the scalibility of the application from a small set of traces.

2) Application specification. An application trace is able to provide considerable detailed information about the dynamic behavior of an application. However the large size of the event log may consume too much memory and storage, which could be a limitation of scalibility. In addition to application trace, the application specification is also utilized to describe the target program for simulation.

Instead of executing the application, researchers describe the applications by the high-level abstractions [127, 42]. Taufer et al. proposed SimBA [127] which is a sequential, discrete event simulator simulating the generation and distribution of tasks running on highly distributed environment. SimBA mimics the BONIC [4] framework for studying the performance of volunteer computing projects. SimBA models the applications as entities, events and resources, and uses finite-state automata to describe the events among the entities in the entire processing to simulate the behavior of tasks. For example, the event represents the condition and the occurrence of an event and results in change of the simulated state.

While formal and compact representation of an application improves the scalibility, the formalism limits the applicability for complex code as it could be too difficult to describe the application logic in an abstraction. To tackle with this problem, some simulators allow users to describe the application in a programmatic way [22, 29, 106, 21]. GridSim [21] is a simulation framework for studying distributed resource management and scheduling for the grid computing environment. It focuses on creating an repeatable and controllable environment for simulation and modeling the performance of the resource management and scheduling algorithms under various user scenarios. GridSim provides an interface, *Gridlet*, for specifying the application-related parameters including length, disk I/O operations and input/output files. This configuration would help the simulator to determine the execution information in order to communicate with other components of the framework. Calheiros et al. [22] proposed CloudSim, a framework for simulating and modeling the infrastructures and the services for cloud computing. CloudSim is built on top of GridSim, and entirely inherits the its user-interface. Therefore CloudSim also supports application description by user-defined specification. SimGrid [29] proposed by Casanova et al. is a simulation toolkit for distributed applications that consumes both application trace and specification for simulation. Specifically, SimGrid provides the interface to utilize the similar but simpler representation of the simulated target which describes an distributed application as a series of communicating concurrent processes.


App Running On BASE system
Instrument binary to capture address stream
Simulate cache hit rates for app on Target system
Each core has summary trace file
core# 1
Running Application
Address stream
Cache Simulator for Target system(s)
core# 2
Running Application
Address stream
Cache Simulator for Target system(s)
core# N
Running Application
Address stream
Cache Simulator for Target system(s)
Address streams generate 44 GB/min (2.6 TB/hour) Per core

Block #	Mem. Ref.	Hit rates
522	1.E08	92,99
523	2.E07	99,100

Block #	Mem. Ref.	Hit rates
521	2.E08	80,85
523	2.E07	97,100

Block #	Mem. Ref.	Hit rates
522	2.E08	99,100
524	3.E07	70,75

Figure 7. Workflow for performance data collection. From [26]

2.4.2 Performance Data Collection

Two major methods employed by simulators for conducting the experiments and collecting the performance metrics are off-line simulation and on-line simulation.

1) Offline simulation. For performing offline simulation, the application is first executed on the real platform. The computation and communication events during the execution will be logged and the configured simulator replays the sequence of events as running the application on the target architecture.

The performance prediction framework [28] consists of two sub-models, the single-processor model and the communication model. It requires the application signatures and machine profiles as the data to generate the both models. Application signatures are collected by tracing. The single-process model needs memory traces collected by MetaSim Tracer [93] while MPIDtrace [99] collects MPI traces for the communication model. The authors use a sampling technique to reduce the trace time, where the tracer samples the memory addresses by a user-defined interval. The sampling results in a partial (percentage depends on the interval) trace. However, it is noted that the total number of memory references estimated for one basic block depends on the sampling percentage. This issue is solved by using two traces. One counts the necessary metric (instructions, floating-point operations) for each basic block. The other one samples the trace of memory access in detail. The two traces technique makes a balance between tracing time and the accuracy of the collected data. Once data is obtained, the last step to build the model is to map the application signatures to the machine profiles of the target architecture by convolution methods [51] implemented in MetaSim Convolver [90] and Dimemas [99] for a single-processor model and communication model respectively. The convolution results show the predicted runtime for an application running on the target machine.

As a trace-based simulator, PSINS [129] requires the execution of the application for collecting the detailed information about MPI events. PSINS utilizes PMPI to enable developers to insert instrumentation code to replace MPI routines at link time for probing the interesting parts. In addition, running the simulation requires parameters to represent the target architecture. Such parameters are available by system measurement and simple obtaining from manufacturers. By configuration of different architectures, PSINS is capable of simulating a wide range of system types, from computational grids to shared-memory multiprocessor architecture.

In [59], the parameters used for the LogGOP model are architecture-related, such as the metric for a per-byte cost of message transmission O, which is collected by precise measurement. Furthermore, the proposed simulator also intercepts all MPI calls to generate the trace information for extrapolation to a larger number of communicators. The data needed for LogGOPSim include architectural parameters and application parameters. Architectural parameters are obtained by measurement which are consumed by the model. Application traces for performing extrapolation is generated by executing the application. By simulating a smaller problem size, LogGOPSim is able to collect and extrapolate the trace information to demonstrate the scaling of the applications.

In 2013, Carrington et al. [26, 25] presented a methodology for characterizing the computation behavior of large-scale application by inferring from the extrapolated results based on the data collected from the executions on a smaller number of core counts. This method avoids the expensive procedure of data collection directly from executions on large number of cores. Instead, it utilizes a smaller number of cores to run the program to obtains data at a lower cost, and extrapolates the data collected for predicting the execution time at large scale. The performance data for modeling the memory behavior falls into two major categories: arithmetic operation and memory operations. The arithmetic operations refer to the floating-point and math operations; and memory operations include load and store references. In order to capture the memory metrics required for modeling, the authors applied PEBIL binary instrumentation platform [73] to instrument every memory access of the application to generate a tracing file which includes the location of the block, the number of floating-point operation, load/store reference counts, size of the memory references and the expected cache hit rates. Then the instrumented binary is executed on a base system to produce the address stream which is then processed by a cache simulator configured according to the memory structures of the target system (Figure 7). The following equation [128] is used for modeling the memory time which accounts for the major time consumption.

$$memory\_time = \sum^{allBBs} (\frac{memory\_ref_{i,j} \times size\_of\_ref}{memory\_BW_i})$$

Another contribution of their work is to extrapolate the data collected from a small number of cores to predict the runtime of a large-scale execution. It provides four canonical functions (constant, linear, exponential and logarithmic) and is used to select the best one which fits the tracing data.

2) Online simulation. In online simulation (direct execution simulation), the applications are directly executed in a controlled environment on the host architecture to obtain dynamic information. The simulator monitors and interacts with the running applications to perform evaluation or other operations, such as intercepting MPI calls.

The LAPSE (Large Application Parallel Simulation Environment) toolkit [35, 36] is a parallel on-line simulator for modeling and predicting the performance of the message-passing parallel program. The inputs for LAPSE includes a descriptive file for building the program, the source code and a LAPSE initialization file for configuring the simulating environment. LAPSE automatically builds the target program and executes it normally. When a communication event is triggered during the execution, it simulates a corresponding communication delay using a simple, pure delay model which ignores the network contention.

Prakash et al. proposed MPI-SIM [110, 5] which is a direct execution-driven parallel simulator for predicting the performance of MPI applications. MPI-SIM predicts the performance of an MPI application as a function of

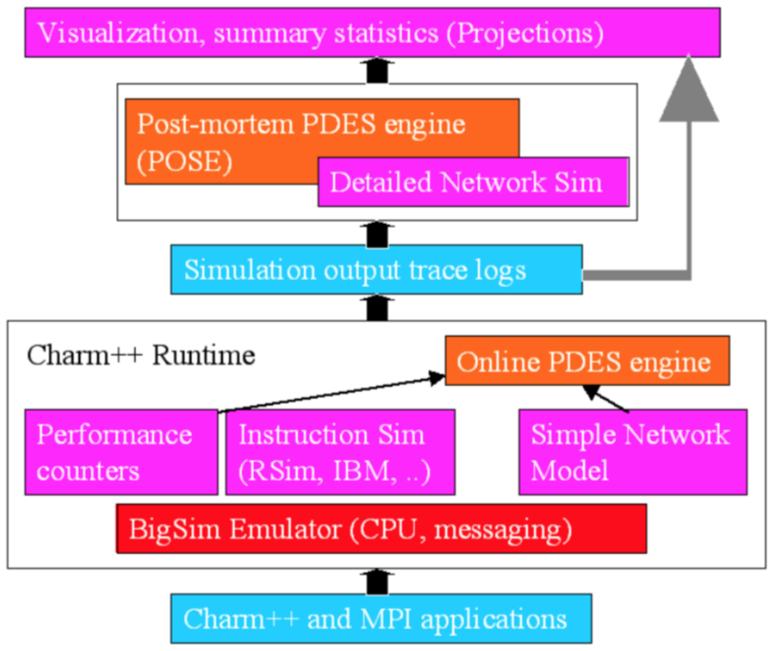

Visualization, summary statistics (Projections)
Post-mortem PDES engine (POSE)
Detailed Network Sim
Simulation output trace logs
Charm++ Runtime
Online PDES engine
Performance counters
Instruction Sim (RSim, IBM, ..)
Simple Network Model
BigSim Emulator (CPU, messaging)
Charm++ and MPI applications

Figure 8. The architecture of BigSim. From [145]

architectural parameters including the number of processors, the underlying network characteristics, and the communication latency. In MPI-SIM, an application thread is divided into two parts depending on whether it requires communication: local code, communication and I/O commands. The local code is directly executed on the host architecture, whereas the communication and I/O operations are intercepted by the simulator to evaluate communication latency and time for I/O operations. To model the communication, MPI-SIM employs both a null message protocol [95] and a conditional event protocol [31]. Compared to the null message protocol, the conditional event protocol is slower, and is only used when the null message protocol fails. The authors also demonstrated that it is safe to completely bypass the simulation protocol for the deterministic code section of the application.

BigSim [145, 146] is another example of execution-driven simulator for predicting the performance of applications on extremely large-scale parallel machines. As an on-line simulator, BigSim shares the similar design ideas as the simulations mentioned above which executes the target program directly. Meanwhile, a parallel algorithm running concurrently interacts with the target to assign the time-stamps for each message. In BigSim, a target program runs on the Charm++ [68] runtime (Figure 8), which makes it easy for the interaction between the target program and the simulation kernel. Charm++ is a parallel programming model which includes parallel objects and object arrays. BigSim is able to evaluate the delay for both sequential (computation) code sections and communication sections. To predict the execution time of the sequential section, it supports three methods: User-supplied description indicates the estimate execution time for each code block on the target machine; the mapping of measured wallclock time from host machine to the target machine by a scale factor; and utilize hardware performance counters for each types of operation on the host architecture and then apply the time measurement of each operation to estimate the total computation time. In addition, the authors introduce POSE to BigSim as a component to perform postmortem simulation. POSE accepts the trace of computation blocks, the messages between them and the dependency information as inputs to replay the computation behavior. POSE aims to address the problem that it has to re-conduct the entire simulation due to the modification of the parameters in the model.

xSim (Extreme-scale Simulator) [15, 41, 40] is an on-line application performance investigation toolkit which was developed in Oak Ridge National Laboratory. xSim is designed as the mid-layer between the MPI application and MPI library. xSim runs the target program directly and intercepts the MPI calls by using the MPI performance tool interface (PMPI). xSim virtualizes the target MPI program as virtual processes which are executed as user-level thread in order to differentiate from native MPI process used for running the simulation itself. This design enables xSim to be enable to achieve a high scalability (134,217,728 simulated MPI ranks [40]).

2.4.3 Advantages and Disadvantages

With the fast development of hardware design and production technology, HPC system become more powerful in computing ability and more efficient in energy consumption and resource management, which brings new opportunities as well as challenges. The increasing complexity of the whole HPC system makes it more difficult to understand behavior and model the performance of arbitrary programs. Simulation-based approaches provide researchers several benefits for accommodating the demand of performance modeling under exascale and complex systems.

Compared with other modeling approaches, simulation-based modeling require less expertise and experience on the architecture and the algorithms. For conducting experiments for arbitrary applications, researchers are not required to master the implementation details of the target application running on particular hardware in order to recognize the abstract relations among the various components describes application behavior. Instead, the experiment procedure can be considered as a "black box" which is transparent to the researchers. Researchers provide the appropriate inputs and collect the necessary data for further processing. This approach saves a huge amount of time for understanding the applications and isolates the performance modeling from development which enables performance modelers to leverage skills on modeling task, and frees developers from analyzing the performance-related data. For occasional cases where the source code of the application is not available for analysis, simulation-based modeling may be an appropriate option. Moreover, simulation-based approaches are resilient to various experiment requirements as the experiment environment is fully controlled by different configurations. Specifically, different types of HPC systems can be simulated in terms of scale, computing power, network topology to satisfy the needs of a particular algorithms. With the appropriate setup, performance of applications on hypothetical architecture can be simulated and studied, which makes it unique to other modeling methods. Another important property of simulation-based approaches is that it is able to provide the precise run-time information of an application. It is possible that an application exhibits different behaviors from the theory because of the running environment. Therefore, the most accurate performance data we could obtain is those collected from actual executions which represent the real CPU load, the memory traffic or the communication pattern, etc. From the perspective of modeling data accuracy, simulation-based approaches stand on top of others.

Simulation-based modeling is able to provide extremely accurate data about the run-time behavior of application, which researchers could use to gain deep insight of the target application. However, to fully understand this modeling technique one should also realize the weaknesses of it. The most critical part of simulation that researchers made efforts to improve is the time and resource efficiency. For example, the full system simulators [88, 16, 113, 142, 84, 114] with complete microprocessor pipeline in software could be able to simulate the flow of instructions through the pipeline and provide accurate information about every cycle. However, the speed of such simulators are much slower than execution of the target program on real platforms due to the complexity of whole system simulation. Therefore, in some scenarios, simulation may not be the best option for exploration of those algorithms running on large-scale system. Because the time consumption of multiple executions for collecting the performance data is not affordable for the researchers and developers who expect the analysis results in a timely fashion in the production environment.

Moreover, resource efficiency is another problem simulation-based modeling approaches cannot ignore. As the scale of the system becomes larger and more complex, the resources required to simulate such system rise up accordingly. It may not a good idea to run simulation-base modeling approaches with limited resource access, which includes the number of CPU, the memory space, the network bandwidth, and the even the power supply. Besides, as other dynamic modeling approaches simulator users may encounter the same problem that the experiment results are inadequate to reflect the intrinsic characteristics of the target applications. It occurs due to the design of inputs that are not able to completely cover all possible code paths which inevitably affect the run-time behavior of applications.

2.5 Summary

In this chapter, we studied several approaches for performance modeling. Analytical methods rely on the human efforts to understand the source code and the interactions between applications and underlying hardware; empirical methods require execution of the target program on the specific architecture to obtain performance metrics and then utilize mathematical approaches to discover the correlation between the input parameters and results; simulation-based methods run the application on the simulator to mimic the behavior of target architecture regardless of its existence to collect performance data.

The criteria we used to distinguish each modeling approach is whether it depends on dynamic collected data and whether the data collection has to be done on the target platform. However, it is noted that the boundaries that separate the approaches are not absolute.

We consider a modeling method as a two-phase procedure: front-end and back-end. The front end collects the dynamically available data while the back end processes them to construct the model. Then we may find that the back-ends of empirical methods such as neural networks and regression also works with the data generated by simulation or compiler. For instance, in [24, 58, 23], the authors proposed semi-empirical modeling approach. It combines analytical modeling with empirical data. To be specific, semi-empirical modeling approach initially builds an analytical model to represent the application and then using empirical data to find the best value for the coefficients to complete the model.

Among the three type of modeling approaches, analytical modeling requires the most human efforts and expertise for analyzing the code. Even for the automatic analytical modeling, human intervention is not avoidable as the annotation is performed manually. However, it is resilient to the changes of the program that the parameters and expressions can be adjusted in a timely fashion, in contrast with empirical and simulation-based methods which require the program to be re-run. Empirical data is the most representative of the behavior of an application, which is likely to improve the accuracy of the modeling. Simulation-based modeling provides more architectural portability and capability of exploring the hypothetical architectures. Both empirical and the simulation-base approaches are able to achieve reasonable automation to arbitrary code.

CHAPTER III

MIRA

This chapter is based on work by Boyana Norris and myself [91]. In this project, Boyana Norris helped with the generalization of the research idea and direction and also provided support for organizing the experiment data and comparison between the results of TAU [117]. I designed and implemented the Mira framework on top of the ROSE [111] compiler framework. Boyana Norris provided insights on selecting the target applications and helped with deriving the results. I prepared the experiment environment for Mira and TAU, performed the experiments, collected the experiment data, and evaluating the results for various inputs.

This chapter describes a fast, accurate, flexible, and user-friendly tool, Mira, for generating performance models by applying static program analysis, targeting scientific applications running on supercomputers. We parse both the source code and binary to estimate performance attributes with better accuracy than considering just source or just binary code. Because our analysis is static, the target program does not need to be executed on the target architecture, which enables users to perform analysis on available machines instead of conducting expensive experiments on potentially expensive resources. Moreover, statically generated models enable performance prediction on nonexistent or unavailable architectures. In addition to flexibility, because model generation time is significantly reduced compared to dynamic analysis approaches, our method is suitable for rapid application performance analysis and improvement. We present empirical validation results to demonstrate the current capabilities of our approach on small benchmarks and a mini-application.

3.1 Motivation

A detailed understanding of application and system performance is critical for effective high-performance software and architecture design. Performance models can provide software developers with useful information about potential bottlenecks and help guide them in identifying optimization opportunities.

As the development of new hardware and architectures progresses, the computing capability of high-performance computing (HPC) systems continues to increase dramatically. However, many applications cannot use the full available computing potential, which wastes a considerable amount of computing power and human effort spent on non-portable optimizations. The inability to fully utilize available computing resources or specific advantages of architectures during application development partially accounts for this waste. Hence, it is important to be able to understand and model program behavior at a fine granularity to gain more information about its bottlenecks and performance potential. Analyzing the instruction mixes of programs at function or loop granularity can provide insight into CPU and memory characteristics, enabling further optimization of a program.

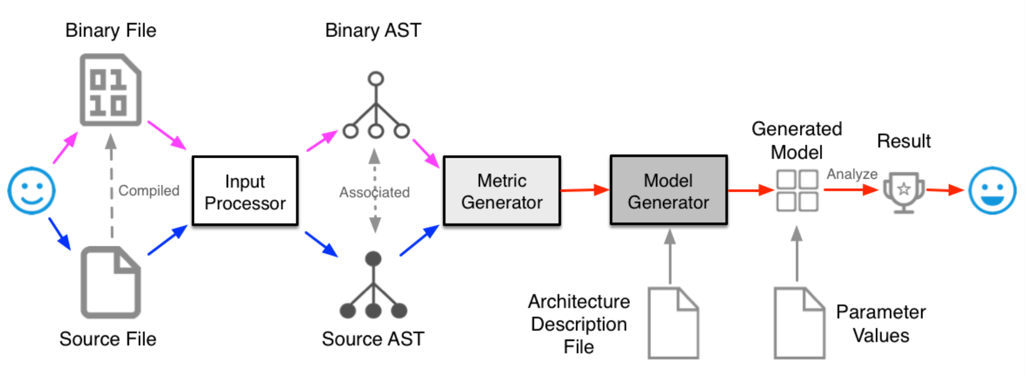


Figure 9. Workflow of Mira for generation of performance model and analysis.

In this chapter, we introduce a new approach for analyzing and modeling programs using primarily static analysis techniques combining both source and binary program information. Our tool, Mira, generates parameterized performance models that can be used to estimate instruction mixes at different granularity (from function to statement level) for different inputs and architectural features without requiring execution of the application.

Current program performance analysis tools can be categorized into static and dynamic. Dynamic (runtime) analysis is performed by executing the target program and measuring metrics of interest, e.g., time or hardware performance counters. By contrast, static analysis operates on the source or binary code without actually executing it. PBound [102] is an example static analysis tool for automatically modeling program performance based on source code analysis of C applications. Because PBound considers only the source code, it cannot capture compiler optimizations and hence produces less accurate estimates of performance metrics.

While some past research efforts mix static and dynamic analysis to create a performance model, relatively little effort has been put into pure static performance analysis and increasing its accuracy. Our approach starts from object code because the code transformations performed by optimizing compilers would cause non-negligible effects on the analysis accuracy. Furthermore, object code is language-independent and more directly reflects runtime behavior. Although object code could provide instruction-level information, it is difficult to manually relate it to the original implementation in a way that enables performance insight directly. For instance, it is difficult or impossible to obtain detailed information about high-level code structures (user-defined types, classes, loops) from just the object code.

Therefore, we also analyze source code to extract this high-level information and combine it with the object code data.

By combining source and object code representations, we can obtain a more precise description of the program and its possible behavior when running on a particular architecture, which results in improved modeling accuracy. The output of our tool can be used to rapidly explore program behavior for different inputs without requiring actual application execution. Moreover, because the analysis is parameterized with respect to the architecture, Mira provides users valuable insight into how programs may run on a particular architecture without requiring access to the actual hardware. Furthermore, the output of Mira can also be applied to create performance models to optimize performance, for example, by enabling static Roofline arithmetic intensity estimates [139].

3.2 Background

This section provides background for the techniques we utilized for the implementation. Mira is built on top of the ROSE [111] framework to support the source and binary code analysis. In addition, we use the polyhedral model to statically describe the loop iteration space.

ROSE Compiler Framework. ROSE [111] is an open-source compiler framework developed at Lawrence Livermore National Laboratory (LLNL). It supports the development of source-to-source program transformation and analysis tools for large-scale Fortran, C, C++, OpenMP and UPC (Unified Parallel C) applications. ROSE uses the EDG (Edison Design Group) parser and OPF (Open Fortran Parser) as the front-ends to parse C/C++ and Fortran. The front-end produces ROSE intermediate representation (IR) that is then converted into an abstract syntax tree (AST). It provides users APIs for program analysis and transformation, such as call graph analysis, control flow analysis, and data flow analysis. The wealth of available analyses makes ROSE an ideal tool both for experienced compiler researchers and tool developers with the minimal background to build custom tools for static analysis, program optimization, and performance analysis.

Polyhedral Model. We rely on the polyhedral model to characterize the iteration spaces of certain types of loops. The polyhedral model is an intuitive parameterized algebraic representation [109] that treats each loop iteration as a lattice point inside the polyhedral space produced by loop bounds and conditions. Nested loops can be translated into a polyhedral representation if and only if they have affine bounds and conditional expressions, and the polyhedral space generated from them forms a convex set. Moreover, the polyhedral model can be used to generate generic representation depending on loop parameters to describe the loop iteration domain. In addition to program transformation [109], the polyhedral model is broadly used for automating optimization and parallelization in compilers (e.g., GLooG [8]) and other tools [47, 17, 49].

3.3 Approach

Mira is built on top of the ROSE compiler framework [111], which provides several useful APIs for parsing source files and disassembling ELF binary files. Mira is implemented in C++ and can process C/C++ source code as input. Figure 9 illustrates the entire workflow of Mira for performance model generation and analysis, which consists of the following three major components:

– **Input Processor**: Input parsing and disassembling;

– **Metric Generator**: AST traversal, metric generation;

– **Model Generator**: Output Python code for models.

Mira provides the architecture description files, although they can be optionally modified by users to allow evaluation on arbitrary architectures of user interest.

3.3.1 Processing Input Files

In this section, we describe in detail our approach to processing the source and corresponding binary source files.

Source code and binary representations. The Input Processor is the front end of Mira; its primary goal is to process source code and ELF object file inputs and build the corresponding abstract syntax trees (ASTs). Mira analyzes these ASTs to locate critical structures such as function bodies, loops, and branches. Furthermore, because the source AST also preserves high-level source information, such as variable names, types, the order of statements, and the right- and left-hand side of the assignment, Mira incorporates this high-level information into the generated model. For instance, one can query all information about the static control part (SCoP) of a loop, including loop initialization, loop condition, and increment (these are not explicit in the binary code). In addition, because variable names are preserved, the identification of loop indices is easier, and the resulting models are more readable and user-friendly.

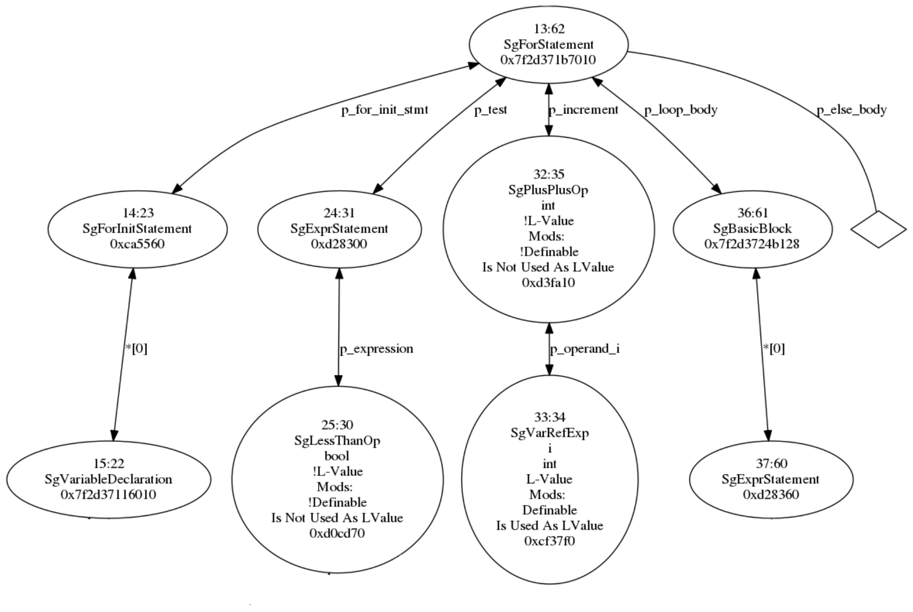


Figure 10. Loop structure from a C++ source code AST fragment (ROSE-generated graph).

Bridge between source and binary. After processing the inputs, two separate ASTs are generated from the source and compiled binary codes representing the structures of the two inputs. Mira uses information retrieved from these trees to improve the accuracy of the generated models. To accomplish this, we build connections between the two ASTs, so that for each structure in the source AST, we can locate corresponding nodes in the binary AST during later traversals.

Although both ASTs are representations of the inputs, they have different shapes, node organizations, and meanings of nodes. A partial binary AST (representing a function) is shown in Figure 11. Each node of the binary AST describes the syntax element of assembly code, such as *SgAsmFunction*, *SgAsmX86Instruction*. As shown in Figure 11, a function in the binary AST is composed of multiple instructions, while in the source AST, a function is composed of statements. Hence, one source AST node typically corresponds to several nodes in the binary AST, which complicates the building of connections between them.

Because the differences between the two AST structures make it difficult to connect the source to binary, an alternative way is needed to make the connection between ASTs more precise. Inspired by debuggers, line numbers are used in our tool as the bridge to associate the source to binary. When we are debugging a program, the debugger knows exactly the source line and column of the error location. By using the *-g* option during program compilation, the compiler will insert debug-related information into the object file for future reference. Most compilers and debuggers use DWARF (debugging with attributed record format) as the debugging file format to organize the information for source-level debugging. DWARF categorizes data into several sections, such as *.debug_info*, *.debug_frame*, etc. The *.debug_line* section stores the line number information.

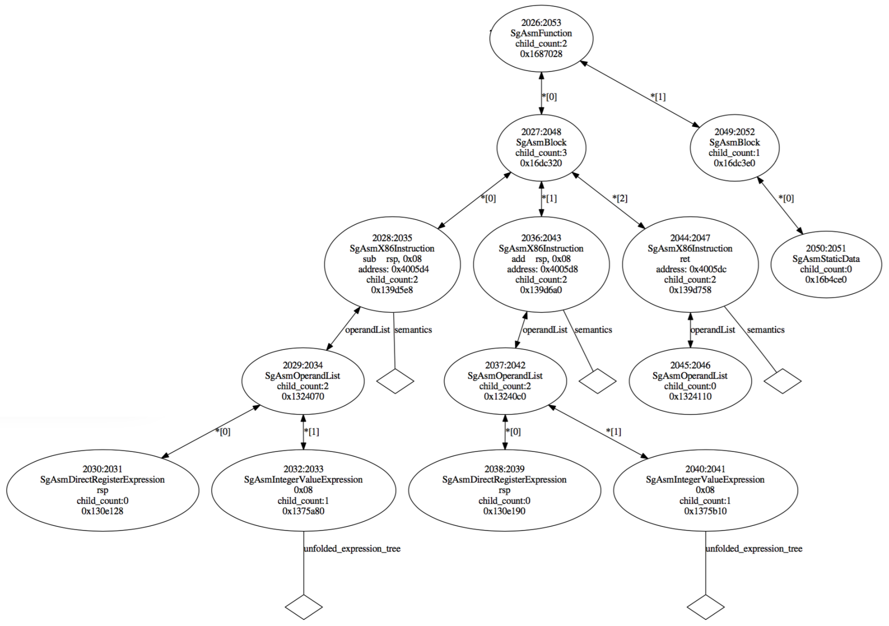


Figure 11. Partial binary AST (ROSE-generated graph).

We use the line number debugging information to decode the specific DWARF section and map the line number to the corresponding instruction address. Because line number information in the source AST is stored by ROSE in each node, unlike in the binary AST, it can be retrieved directly. After line numbers are obtained from both source and binary, connections are built in each direction between the two ASTs. As mentioned in the previous section, a source AST node normally links to several binary AST nodes due to the different meanings of nodes. Specifically, a statement contains several instructions, but an instruction only has one connected source location. Once the node in the binary AST is associated with the source location, further analysis can be performed. For instance, it is possible to narrow the analysis to a small scope and collect data such as the instruction count and type in a particular code fragment, such as function body, loop body, and even a single statement.

3.3.2 Generating Metrics

The metric generator is an important part of the entire framework, which has a significant impact on the accuracy of the generated model. It receives the ASTs as inputs from the Input Processor to produce metrics for model generation. An AST traversal is needed to collect and propagate necessary information about the specific structures in the program for appropriate organization of the program representation to precisely guide model generation. During the AST traversal, additional information is attached to the particular tree node as attributes and later used for analysis and modeling. For example, if a statement is very long, it is probably split across several lines. In this case, all the line numbers are collected together and stored as extra information attached to the statement node.

In order to gather instruction statistics, the metric generator traverses the source AST twice: first bottom-up and then top-down. The upward traversal propagates detailed information about specific structures up to the head node of the subtree. For instance, as shown in Figure 10, *SgForStatement* is the head node of the loop subtree; however, this node itself does not store any information about the loop. Instead, the loop information such as loop initialization, loop condition and step are stored in *SgForInitStatement*, *SgExprStatement* and *SgPlusPlusOp* separately as child nodes. In this case, the bottom-up traversal recursively collects information from leaves to root and organizes it as extra data attached to the head node for the loop. The attached information will serve as the context in modeling.

After bottom-up traversal, top-down traversal is applied to the AST. Because information about the subtree structure has been collected and attached, the downward traversal primarily focuses on the head node of the subtree and other nodes of interest, for example, the *loop* head node, *if* head node, *function* head node, and *assignment* node, etc. Moreover, the top-down traversal must pass down necessary information from the parent to the child node in order to model complicated structures correctly. For example, when a loop contains a branch or a nested loop, the inner structure requires the information from the parent node as the outer context to produce the parameterized metric expressions. Finally, instruction information from the ELF AST is connected and associated with corresponding structures in the source AST during the top-down traversal.

3.3.3 Generating Models

The model generator consumes the intermediate analysis result produced by the metric generator and generates an easy-to-use model. For the greatest flexibility, the generated model is in Python so that the result of the model can be applied to various scientific libraries for further analysis and visualization. Note that generating models in a different language can be accomplished by creating a different Model Generator implementation while the other Mira components remain unchanged. In some cases, the model is in ready-to-execute condition for which users can run it directly without providing any input. However, users are required to provide extra input to run the model when the model contains parametric expressions that depend on values, which are not known at compile time. For example, when user input is expected in the source code, or the value of a variable is returned in a virtual function call, the variable names are preserved in the model as parameters that will be specified by the users before running the model.

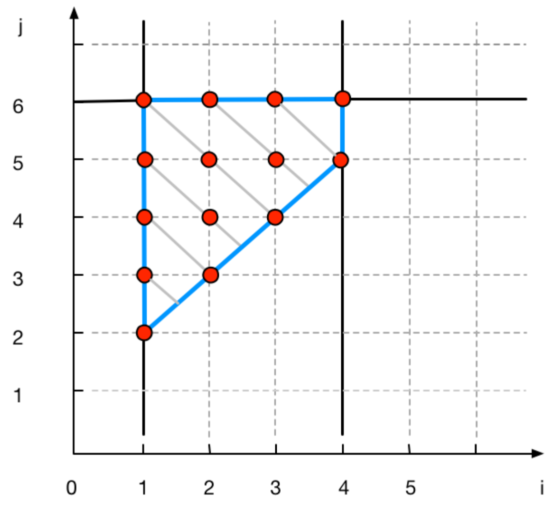


Figure 12. Polyhedral representation of a nested loop.

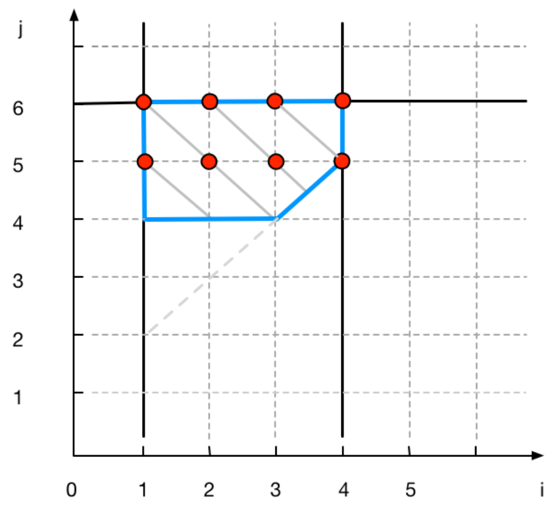


Figure 13. Polyhedral representation with the *if* constraint.

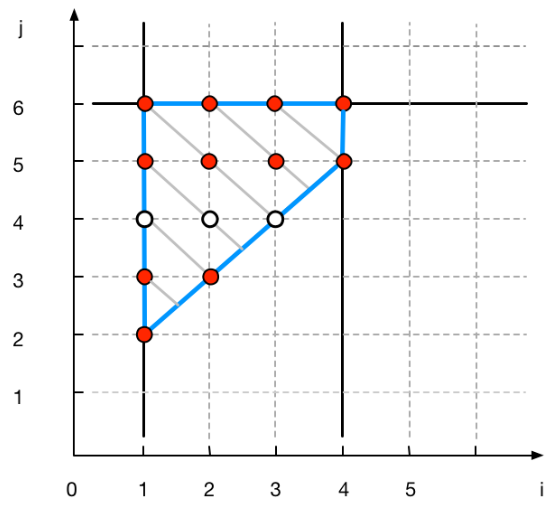


Figure 14. The *if* constraint causing holes in the polytope.

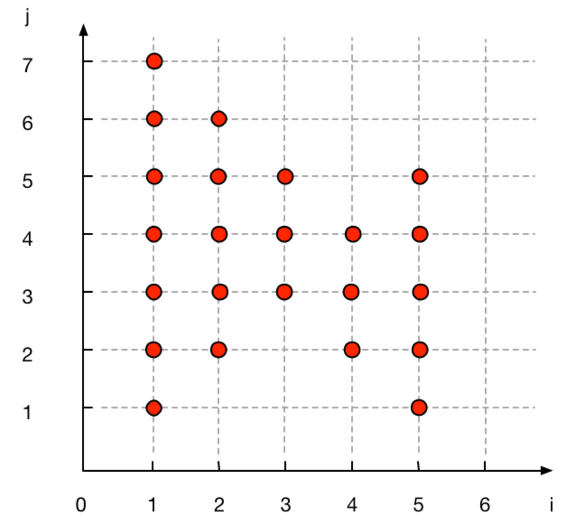


Figure 15. Exceptions in polyhedral modeling.

Loop modeling. Loops are common in HPC codes and are typically at the heart of the most time-consuming computations. A loop executes a block of code repeatedly until certain conditions are satisfied. Bastoul et al. [9] surveyed multiple high-performance applications and summarized the results in Table 1. The first column shows the number of loops contained in the application. The second column lists the total number of statements in the applications, and the third column counts the number of statements covered by loop scope. The ratio of in-loop statements to the total number of statements is calculated in the last column. In the data shown in the table, the lowest loop coverage is 77% for *quake*, and the coverage rates for the rest of the applications are above 80%. This survey data also indicates that the in-loop statements make up a large majority portion of the total statements in the selected high-performance applications.

Using the polyhedral model. Whether loops can be precisely described and modeled or not has a direct impact on the accuracy of the generated model because the information about loops will be provided as context for the analysis

Table 1. Loop coverage in high-performance applications.

Application	Num. of loops	Num. of statements	Statements in loops	Percentage
applu	19	757	633	84%
apsi	80	2192	1839	84%
mdg	17	530	464	88%
lucas	4	2070	2050	99%
mgrid	12	369	369	100%
quake	20	639	489	77%
swim	6	123	123	100%
adm	80	2260	1899	84%
dyfesm	75	1497	1280	86%
mg3d	39	1442	1242	86%

of statements inside the loops. The term "loop modeling" refers to the analysis of the static control parts (SCoP) of a loop to obtain the information about the loop iteration domain, which includes the understanding of the initialization, termination condition, and step. Unlike dynamic analysis tools that collect runtime information during execution, our static approach primarily relies on SCoP parsing and analyzing for loop modeling. To model a loop, we take several factors into consideration, such as depth, data dependencies, bounds, etc. Listing 3.1 shows a basic loop structure, the SCoP is complete and simple without any unknown variables.

Listing 3.1 Basic loop example.

```
for (i = 0; i < 10; i++) {
      statements;
   }
```

For this case, it is possible to retrieve the initial value, upper bound, and increment expression from the AST, then calculate the number of iterations. The iteration count is used as context when analyzing the loop body. For example, if corresponding instructions are obtained from the binary AST for the statements in Listing 3.1, the actual count of these instructions is expected to be multiplied by the iteration count to describe the real situation during runtime.

Listing 3.2 Double-nested loop example.

```
for(i = 1; i <= 4; i++)
   for(j = i + 1; j <= 6; j++) {
    statements;
  }
```

Loops in applications are usually more complicated than the simple example above, and we must handle as many special cases as possible. In Mira, first, we use the polyhedral model to accurately model loops whenever they fit the constraints of the polyhedral approach. In some cases, the index of inner loop has a dependency on the outer loop index. As shown in Listing 3.2, the initial value of the inner index j is based on the value of the outer index i. For this case, it is possible to derive a formula as the mathematical model to represent this loop, but it would be difficult and timeconsuming. Most importantly, it is not general; the derived formula may not fit for other scenarios. To use the polyhedral model for this loop, the first step is to represent loop bounds in affine functions. The bounds for the outer and inner loop are $1 \le i \le 4$ and $i + 1 \le j \le 6$, which can be written as two equations separately:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 4 \end{bmatrix} \ge 0$$

$$\begin{bmatrix} -1 & 1 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 6 \end{bmatrix} \geq 0$$

In Figure 12, the two-dimensional polyhedral area representing the loop iteration domain is created based on the two linear equations. Each dot in the figure represents a pair of loop indexes (i, j), which corresponds to one iteration of the loop. Therefore, by counting the integers in the polyhedral space, we can parse the loop iteration domain and obtain the iteration times. For loops with more complicated SCoP, such as the ones that contain variables instead of concrete numerical values, the polyhedral model is also applicable. When modeling loops with unknown variables, Mira uses the polyhedral model to generate a parametric expression representing the iteration domain, which can be changed by specifying different values to the input. Mira maintains the generated parametric expressions and uses them as the context in the following analysis. In addition, the unknown variables in loop SCoP are preserved as parameters until the parametric model is generated. With the parametric model, it is not necessary for the users to regenerate the model for different values of the parameters. Instead, they just have to adjust the inputs for the model and run the Python code to produce a concrete value.

Listing 3.3 Exception in polyhedral modeling.

```
for (i = 1; i <= 5; i++)
  for (j = min(6 - i, 3);
         j <= max(8 - i, i); j++) {
    statements;
  }
```

There are exceptions where the polyhedral analysis cannot be applied. For the code snippet in Listing 3.3, the SCoP of the loop forms a non-convex set (Figure 15), which is not handled by the polyhedral model. Another problem in this code is that the loop initial value and loop bound depend on the return values of function calls. For static analysis to track and obtain such values, a more complex interprocedural analysis is required, which we plan to do in future work.

Listing 3.4 Loop with *if* constraint

```
for (i = 1; i <= 4; i++)
   for (j = i + 1; j <= 6; j++) {
     if (j > 4) {
       statements;
     }
   }
```

Branches. In scientific applications, branch statements are frequently used in loops, for example, to handle boundary conditions or detect convergence. In Listing 3.4, the *if* constraint $j > 4$ is introduced into the previous code snippet. The number of execution times of the statement inside the *if* depends on the branch condition. In our analysis, the polyhedral model of a loop is kept and passed down to the inner scope. Thus the *if* node has the information of its outer scope. Because the loop conditions combined with branch conditions form a polyhedral space as well, shown in Figure 13, the polyhedral representation is still able to model this scenario by adding the branch constraint and regenerate a new polyhedral model for the *if* node. Comparing Figure 13 with Figure 12, it is obvious that the iteration domain becomes smaller and the number of integers decreases after introducing the constraint, which indicates the execution times of statements in the branch is limited by the *if* condition.

Listing 3.5 *if* constraint breaks polyhedral space

```
for (i = 1; i <= 4; i++)
  for (j = i + 1; j <= 6; j++) {
    if (j % 4 != 0) {
      statements;
    }
  }
```

However, some branch constraints might break the definition of a convex set so that the polyhedral model is not applicable. For the code in Listing 3.5, the *if* condition excludes several integers in the polyhedral space causing “holes” in the iteration space as shown in Figure 14. The excluded integers in the true branch break the integrity of the space so that it no longer satisfies the definition of the convex set; hence the polyhedral method cannot be used. Nevertheless, the false branch still satisfies the polyhedral model, so we can solve the following equation to determine the true branch values:

$$Count_{true\_branch} = Count_{loop\_total} - Count_{false\_branch}$$

The generality of the polyhedral model makes it suitable for most common cases in real applications; however, there are some cases that cannot be handled by the polyhedral model or any static analysis. For such circumstances, we provide users an option to annotate branches or the loops which Mira cannot handle statically.

Annotation. There are loop and branch cases that we are not able to process statically, such as conditionals involving loop index-unrelated variables or external function calls used for computing loop initial values or loop/branch conditions. Mira accepts user annotations to address such problems. We designed three types of annotation: an estimated percentage or a numerical value representing the proportion of iterations branch may take place inside the loop or the number of iterations, which simplifies the loop/branch modeling; a variable used as initial value or condition to complete the polyhedral model, or a flag to indicate that a structure or a scope should be skipped. To annotate the code, users just need to put the information in a "#pragma" directive in this format: *#pragma @Annotation information.* Mira processes the annotations during metric generation.

Listing 3.6 User annotation for *if* statement.

```
for (i = 1; i <= 4; i++)
   for (j = a[i]; j <= a[i+6]; j++) {
   #pragma @Annotation \
          {lp_init:x,lp_cond:y}
    if (foo(i) > 10) {
      #pragma @Annotation {skip:yes}
      statements;
    }
  }
```

As the example shown in Listing 3.6, the *if* has a function call as a condition that causes a failure when Mira tries to generate the model fully automatically. To solve this problem, we specify an annotation in the pragma to provide the missing information and enable Mira to generate a complete model. In the given example, the whole branch scope will be skipped when generating metrics. Besides, we also annotate the initial value and condition of the inner loop using variable x and y because, as a static tool, Mira is not able to obtain values from those arrays. Mira will use the two variables to complete the polyhedral model; these variables will be treated as parameters expecting sample values from the user at model evaluation time.

Functions. Mira organizes the generated model in functions, which correspond to functions in the source code. In the generated model, the function header is modified for two reasons: flexibility and usability. Specifically, each user-defined function in the source code is modeled into a corresponding Python function with a different function signature, which only includes the arguments that are used by the model. In addition, the generated model function has a slightly different name in order to avoid potential conflict due to different calling contexts or function overloading. For instance, the Python function with name *foo_2* represents the original C++ function *foo*, but with a reduced number of arguments. In the body of *foo_2*, the original C++ statements are replaced with corresponding instruction counter metrics retrieved from binary. These data are stored in Python dictionaries and updated in the same order as the original statements. Each function, when called, returns the aggregate counts within its scope. This design enables users to (optionally) separate and obtain the overview of particular functions with only minor changes to the model.

Correct handling of function calls involves two aspects: locating the corresponding function and combining the metrics into the caller function. To combine the metrics, we designed a Python helper function *handle_function_call*, which takes three arguments: caller metrics, callee metrics, and loop iterations. It enables Mira to model the function call in the loop, in which each metric of the callee is multiplied by the number of loop iterations. Mira retrieves the name of the callee function from the source AST node and then generates a function call statement in Python and takes the return values that representing the metrics in the callee function. After that, Mira calls the *handle_function_call* to combine metrics of the caller and the callee.

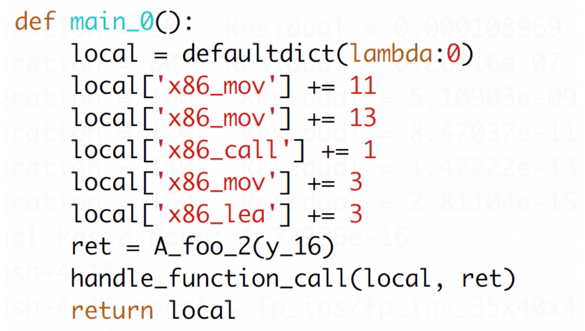
```
class A{
public:
    void foo(double *a, double *b){
        for(int i = 0; i < 10; i++)
            for(int j = 0; j < b[i]; j++){
            #pragma @Annotation {lp_cond:y}
            a[j] = a[j] * b[j];
        }
    }
};

int main(){
    A a;
    double m[] = {1.0, 2.0, 3.0, 4.0, 5.0};
    double n[] = {5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
    a.foo(m, n);
}
```

Figure 16. Original source code.

```
def A_foo_2(y):
    local = defaultdict(lambda:0)
    local['x86_cmp'] += 1 * 10
    local['x86_add'] += 1 * 10
    local['x86_jmp'] += 1 * 10
# variables [y]
    if y > 0:
        local['x86_cvtsi2sd'] += 1 * 10 * y
    # NOT ENTIRE GENERATED FUNCTION
    # ...
# variables [y]
    if y > 0:
        local['x86_add'] += 2 * 10 * y
# variables [y]
    if y > 0:
        local['x86_mulsd'] += 1 * 10 * y
    return local
```

Figure 17. Python model code excerpt for the *foo* function.



```
def main_0():
    local = defaultdict(lambda:0)
    local['x86_mov'] += 11
    local['x86_mov'] += 13
    local['x86_call'] += 1
    local['x86_mov'] += 3
    local['x86_lea'] += 3
    ret = A_foo_2(y_16)
    handle_function_call(local, ret)
    return local
```

Figure 18. Python model code for the main program.

Architecture description file. To enable the evaluation of the generated performance model in the context of specific architectural features, we provide an architecture description file where users define architecture-related parameters, such as the number of CPU cores, cache line size, and vector length. Moreover, this user-customizable[1] file can be extended to include the information which does not exist in the source or binary file to enable Mira to generate a complete model. For example, we divided the x86 instruction set into 64 different categories in the description file, which Mira uses to estimate the number of instructions in each category for each function in the source file. This representation strikes a balance between fine and coarse-grained approaches, providing category-based cumulative instruction counts at fine code granularity (down to statement-level) and enabling a better understanding of local behavior. Based on the metrics Mira generated in Table 2, Figure 19 illustrates the distribution of categorized instructions from function *cg_solve* from the miniFE application [56]. The separated piece represents the SSE2 vector instructions, which are the source of the floating-point instruction in this function.

[1]Note that users are not required to read or modify this file at all to use Mira effectively.

Table 2. Categorized instruction counts of function `cg_solve`.

Category	Count
Integer arithmetic instruction	6.8E8
Integer control transfer instruction	2.26E8
Integer data transfer instruction	2.42E9
SSE2 data movement instruction	3.67E8
SSE2 packed arithmetic instruction	1.93E8
Misc Instruction	2.77E8
64-bit mode instruction	2.59E8

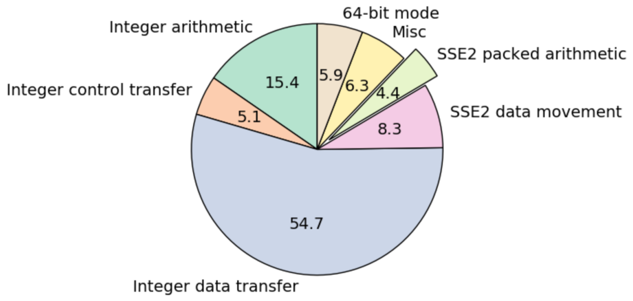


Figure 19. Instruction distribution of function cg_solve.

Generated model. We describe the model generated (output) by Mira with an example that shows the source code (input) and generated Python model separately. The source code (Figure 16) includes a class *A* defining a member function *foo* with two array variables as the parameters. The member function *foo* is composed of a nested loop in which we annotate the upper bound of the inner loop with variable *y*. In the *main* function, Mira creates an instance of class *A* and call function *foo*. Figure 17 shows part of the generated Python function *foo* in which the new function name is replaced with the combination of its class name, original function name, and the number of arguments in the original function definition. The body of the generated function *A_foo_2* consists of the Python statements for keeping track of performance metrics. Mira uses the annotation variable *y* to complete the polyhedral model and preserves *y* as the argument.

Table 3. FPI counts in the STREAM benchmark.

Array size / Tool	TAU	Mira	Error
2M	8.239E7	8.20E7	0.47%
50M	4.108E9	4.100E9	0.19%
100M	2.055E10	2.050E10	0.24%

Table 4. FPI counts in DGEMM benchmark.

Matrix size / Tool	TAU	Mira	Error
256	1.013E9	1.0125E9	0.05%
512	8.077E9	8.0769E9	0.0012%
1024	6.452E10	6.4519E10	0.0015%

Similarly, the generated function *main* is shown in Figure 18. It calls the *A_foo_2* function and then updates its metrics by invoking *handle_function_call*. The parameter *y_16* indicates that the function call associates the source code at line 16. At present, the value of *y_16* is specified by users during model evaluation. Different values can be supplied as function parameters in different function call contexts.

3.4 Evaluation

In this section, we evaluate the correctness of the model derived by Mira with TAU in the instrumentation mode. Two benchmarks are separately executed statically and dynamically on two different machines. While Mira counts all types of instructions, we focus on floating-point instructions (FPI) in this section because it is an important metric for HPC code analysis, which is not obtainable through measurement on some recent Intel processors. The validation is performed by comparing the floating-point instruction counts produced by Mira with empirical instrumentation-based TAU/PAPI measurements.

3.4.1 Experiment Environment

We conducted the validation on two machines:

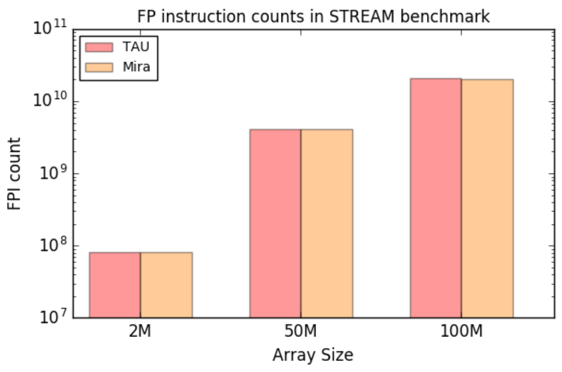
- **Arya**: two Intel Xeon E5-2699v3 2.30GHz 18-core Haswell CPUs and 256GB of memory.
- **Frankenstein**: two Intel Xeon E5620 2.40GHz 4-core Nehalem CPUs and 22GB of memory.

3.4.2 Benchmarks

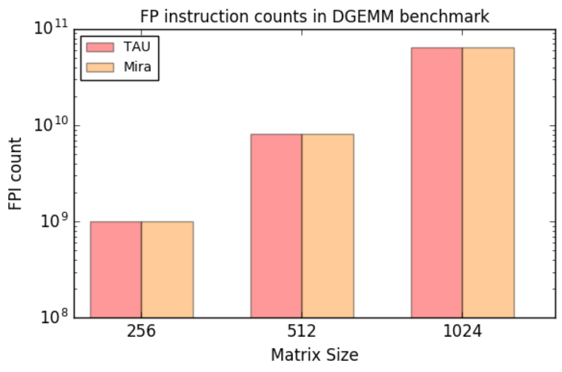
First, we validate Mira-generated models with two simple benchmarks: STREAM [89] and DGEMM [87]. STREAM is designed for the measurement of sustainable memory bandwidth and corresponded computation rate for simple vector kernels. DEGMM is a widely used benchmark for measuring the floating-point rate on a single CPU. It uses double-precision real matrix-matrix multiplication to calculate the floating-point rate. For both benchmarks, the non-OpenMP version is selected and executed serially with one thread.

Table 5. FPI Counts in miniFE

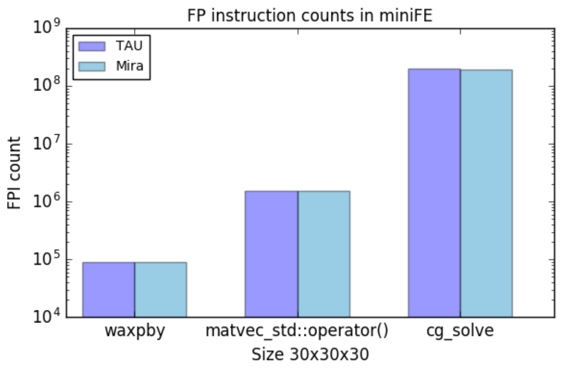
size	Function / Tool	TAU	Mira	Error
30x30x30	waxpby	8.95E4	8.94E4	0.011%
	matvec_std::operator()	1.54E6	1.52E6	1.3%
	cg_solve	1.966E8	1.925E8	2.09%
35x40x45	waxpby	2.039E5	2.037E5	0.098%
	matvec_std::operator()	3.57E6	3.46E6	3.08%
	cg_solve	7.621E8	7.386E8	3.08%



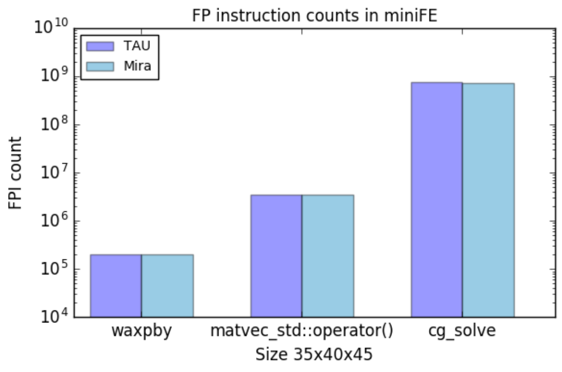
(a) FP instruction counts in STREAM benchmark



(b) FP instruction counts in DGEMM benchmark



(c) FP instruction counts in miniFE



(d) FP instruction counts in miniFE

Figure 20. Validation of floating-point instruction counts.

3.4.3 Mini Application

In addition to the STREAM and DGEMM benchmarks, we also use the miniFE mini-application [56] to verify the result of Mira. MiniFE is composed of several finite-element kernels, including computation of element operators, assembly, sparse matrix-vector product, and vector operations. It assembles a sparse linear system and then solves it using a simple unpreconditioned conjugate-gradient algorithm. Unlike STREAM and DGEMM, in which the *main* function takes the majority part of the code, miniFE contains several functions and includes function calls which, challenge the capability of Mira to handle a long chain of function calls.

3.4.4 Results

In this section, we present empirical validation results and illustrate the tradeoffs between static and dynamic methods for performance analysis and modeling. We also show a use case for the generated instruction metrics to compute an instruction-based arithmetic intensity derived metric, which can be used to identify loops that are good candidates for different types of optimizations (e.g., parallelization or memory-related tuning).

1) Discussion. Tables 3, 4 and 5 show floating-point instruction counts in two benchmarks and mini application separately. The metrics are gathered by evaluating the model generated by Mira and comparing it to the empirical results obtained through instrumentation-based measurement using TAU and PAPI.

In Figure a, the X-axis is the size of the input array, and we choose 20 million, 50 million, and 100 million, respectively. The logarithmic Y-axis shows floating-point instruction counts. Similarly, in Figure b, the X-axis is for input size and the Y for FPI counts. Figure c and Figure d show FPI counts for three functions for the different problem sizes. We show details for the *cg_solve* function, which solves the sparse linear system because it accounts for the bulk of the floating-point computations in this mini-app. The function *waxpby* and the operator overloading function *matvec_std::operator()* are in *cg_solve's* call tree and are invoked in the loop. Our results show that the floating-point instruction counts produced by Mira are close to the TAU measurements (of the PAPI_FP_INS values), with an error of up to 3.08%. The difference between static estimates and measured quantities increases with problem size, which means that there are discrepancies within some of the loops. This is not unexpected—static analysis cannot capture dynamic behavior with complete accuracy. The measured values capture samples based on all instructions, including those in external library function calls, which at present are not visible and hence not analyzed by Mira. For such scenarios, Mira can only track the function call statements that just contain several stack manipulation instructions while the content of the invoked function is skipped. In the future, we plan to provide different mechanisms for handling these cases, including limited binary analysis of the corresponding library supplemented by user annotations.

In addition to correctness, we compare the execution time of the static and empirical approaches. In empirical approaches, the experiment has to be repeated for different input values, and in some cases, multiple runs for each input value are required (e.g., when collecting performance hardware counter data). Instrumentation approaches can focus on specific code regions, but most sampling-based approaches collect information for all instructions; hence they potentially incur runtime and memory cost for collecting data on uninteresting instructions.

By contrast, our model only needs to be generated once and then can be evaluated (at low computational cost) for different user inputs and specific portions of the computation. Most important, the performance analysis by a parametric model can be used to achieve broad coverage without incurring the costs of many application executions.

Another challenge in dynamic approaches is the differences in hardware performance counters, including the lack of availability of some types of measurements. For example, in modern Intel Haswell servers, there is no support for FLOP or FPI performance hardware counters. Hence, static performance analysis may be the only way to produce floating-point-based metrics in such cases.

2) Prediction. Next, we demonstrate how one can use the Mira-generated metrics to model the estimated instruction-based floating-point arithmetic intensity of the *cg_solve* function. The general definition of arithmetic intensity is the ratio of arithmetic operation to memory traffic. With an appropriate setting in the architecture description file, we can enable Mira to generate various metrics. As the data shown in Table 2, Mira categorizes the instructions in *cg_solve* into seven categories. In the listed categories, "SSE2 packed arithmetic instruction" represents the packed and scalar double-precision floating-point instructions and "SSE2 data movement instruction" describes the movement of double-precision floating-point data between XMM registers and memory. Therefore the instruction-based floating-point arithmetic intensity of function *cg_solve* can be simply calculated as $1.93E8/3.67E8 = 0.53$. This is a simple example to demonstrate the usage of our model. With more complex architecture description files, Mira is able to perform more complicated predictions.

3.5 Summary

This chapter presents the Mira framework. It combines the binary and source code static analysis to automatically construct performance models, which affirmatively answers RQ1 ("Can we use static binary analysis to construct representations of programs for performance modeling?"). The parameterized representation of the program generated from the combination of binary and source code improves the accuracy and the readability of the model comparing to solely relying on either binary or source code. With our evaluation with the state-of-the-art performance analysis tool (TAU) using floating-point operations hardware counters, we demonstrated that the Mira-generated model accuracy is excellent. The error in the model predictions is below 0.5% for the benchmarks and 3% for the MiniFE application.

CHAPTER IV

MELIORA

This chapter is based on the work [92] by Boyana Norris and myself. In this research, Boyana Norris helped with the generalization of the research idea and direction and also provided support for organizing the experiment data and comparison between the results of autotuning. I designed and implemented the Meliora framework. Boyana Norris provided insights on refining the model and helped with deriving the results.

4.1 Motivation

Performance models are useful not only in the initial implementation of algorithms but for the entire software life-cycle. For example, a performance model can guide the application developers in choosing the most suitable algorithm to reach the design goal. In the phase of application testing, it offers direct feedback on the performance and resource consumption of an application by adjusting parameters. In addition to software development, performance modeling also plays an important role in system design and tuning. Specifically, performance models are capable of describing non-existent hardware, which helps researchers to explore their options when designing new systems. For example, the researchers may model several potential network topologies to select the most efficient way to connect nodes in a large-scale parallel system. A performance model can be used to predict the expected design performance of a platform with a given configuration. The expectation can be compared with the benchmarked performance to reveal tuning opportunities. Thus, performance modeling is significantly useful for both software and hardware development.

Traditional approaches to model-based performance code optimization employ a variety of techniques to represent the behavior of existing implementations. Analytical models are more difficult to create but can also be used to represent the performance of potential implementations that cannot be measured directly. Increasingly, machine learning (ML) methods are being used for performance modeling to identify patterns in the code feature space and identify optimal parameters. They are an attractive option because of the potential to capture complex patterns mostly automatically.

The main goal of our research is to provide a novel, scalable ML-based code matching methodology, and tool that enables the accurate identification of loop-based computations. At present, when faced with a new code to optimize, most humans, compilers, and autotuners start from scratch or, at best, apply some general heuristics when deciding on what code transformations to pursue. What we propose in this paper is a methodology to provide a high-quality starting point for the optimization of any computation. When used by humans, this can save hours, days, or weeks of effort. When used in conjunction with compilers or autotuners, this approach can result in performance that is competitive with that of empirically tuned codes but at a small fraction of the cost.

Our primary objective is to accelerate the code optimization process by using a deep learning technique to match a target code to similar computations that have been optimized previously. To accomplish this, we define a new graph-based code representation and combine it with a code generation framework to enable the automated creation of a deep learning model for matching loop-based computations. We name this approach Meliora, which translated from Latin means "ever better". The approach is based on learning accurate graph embeddings of a set of computational kernels $K_0, K_1, ..., K_N$ that have been autotuned or manually optimized on the target architecture. When a new code C_{new} must be considered, we apply the model to identify which optimized kernel, K_i, is the closest match. Based on that information and the autotuning results, we can select the best-performing version of K_i, $K_{i_{opt}}$, from our training set. This information can then be used by a human developer to manually optimize their implementation (which may involve significant refactoring), or it can be used by a compiler or an autotuner to automatically apply a small set of optimizations. The Meliora framework can thus greatly reduce or eliminate the exponential search space of potential optimizations.

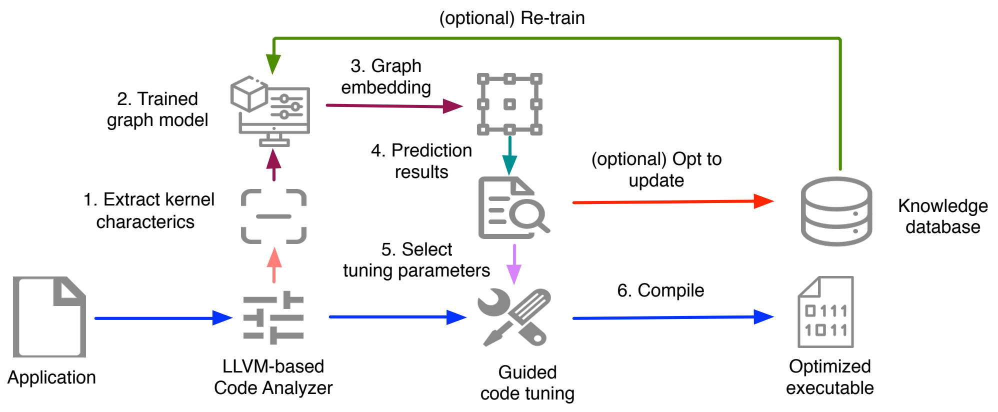


Figure 21. Meliora's workflow for code representation-based model generation and optimization.

The two principal components of the Meliora approach are an LLVM-based frontend for extracting code features and an ML-based graph embedding component for learning efficient code representations. The overall workflow of feature extraction, model generation, and subsequent code optimization is illustrated in Figure 21.

The research contributions in this chapter aim to answer **RQ2** and can be summarized as follows.

– Definition of a graph-based code representation that extends the traditional control flow graph with computationally relevant features, such as instruction mix and reuse-distance data.

– A deep-learning framework for learning code graph embeddings for efficient and accurate matching of computational kernels composed of code that does not contain non-standard function calls.

– Integration with an autotuner to enable fully automated construction of the training dataset for a supervised machine learning model used for matching the code graph embeddings.

– Evaluation of the accuracy and effectiveness of this approach on a set of computational kernels from the SPAPT benchmark suite.

This chapter is organized as follows: Section 4.2 briefly describes the graph representation learning techniques, the LLVM compiler framework, and the background of performance optimization approaches. In Sections 4.3, we discuss the details of our methodology and the implementation. Section 4.4 evaluates the efficiency of Meliora in metric retrieval and the accuracy of the generated models on several scientific kernels. In Section 4.5, we introduce related work about performance optimization and graph representation learning. Section 4.6 concludes with a summary and future work discussion.

4.2 Background

Before discussing Meliora in detail, we briefly overview the concepts and tools on which we have based our approach.

LLVM Meliora's static analysis component is based on the LLVM compiler framework [72], which contains a set of open-source tools and libraries for writing code analysis and transformation tools for several languages and most HPC architectures. For example, the *Clang* compiler frontend provides several useful static analyses and APIs to manipulate the Abstract Syntax Trees (ASTs) generated from C and C++ source code. *LLDB* is the LLVM version of GDB for efficient debugging. *Polly* applies the polyhedral model [109], an intuitive parameterized algebraic representation, to optimize the memory access pattern within loops. The middle layer, LLVM intermediate representation (IR), connects the frontend and the backend of a program and, at the same time, offers a language-independent environment for developing new, portable tools, thereby reducing frontend development efforts. The metric generation components of Meliora (Sections 4.3.2 and 4.3.3) are hence built on top of the LLVM IR for compatibility with code written in different programming languages.

Graph Representation Learning Graph representation learning (or graph embedding) is a type of machine learning that focuses on creating compact vector representations of graphs. To be specific, for a given graph G its goal is to find a mapping $f : v_i \rightarrow x_i \in \mathbb{R}^d$, such that the embedded vector $X_i = x_1, x_2, ..., x_d$ can represent the properties of the original graph. There are four general types of embeddings: node, edge, hybrid, and whole-graph embedding. Algorithms that operate on embeddings rather than graphs can greatly reduce the cost of computation and storage; hence embeddings are widely used in various types of applications, such as link prediction, node classification in social networks, and DNA analysis in computational biology. The challenge is to balance the compactness with the preservation of key properties. Methods for graph representation learning include those based on *dimensionality reduction*, which employs mathematical analysis techniques, such as principal component analysis (PCA) [66], linear discriminant analysis (LDA) [141], and locally linear embedding (LLE) [115]. Techniques include *matrix-factorization* [118, 11] are also used for graph learning. *Random walk* is utilized by graph embedding approaches like DeepWalk [108] and Node2Vec [50], which samples a graph with many paths to find the context of the connected vertices. More recently, researchers have also employed convolutional neural networks (CNNs) [75, 74] and recurrent neural networks (RNNs) [94] to learn graph representations, for example, in GraphSAGE [53]. We take a CNN-based approach to support code graph embedding because it is less limited than earlier approaches (e.g., we must handle directed graphs with node and edge labels).

Performance Optimization One of the common approaches to optimizing performance is automatic performance tuning (autotuning). Here we consider autotuning that not only explores simple parameters but rather considers significantly different code variants (e.g., through transformations such as tiling or automatic parallelization). Orio [55], the autotuning framework we used to build the training dataset is based on annotated C or Fortran code, coupled with a tuning specification containing various transformations and their parameters. In general, Meliora does not require the use of an autotuner; however, it enables the relatively easy generation of a sizeable training dataset for our model.

4.3 Approach

Meliora is a novel framework for characterizing computational kernels whose goal is to dramatically reduce the time and effort required for optimizing performance. The primary objective is to accelerate the process of searching the space of optimizations by using a CNN-based technique to identify previously optimized similar codes. When used in conjunction with an autotuner, Meliora can greatly reduce or eliminate the exponential search space of parameterized code versions. Here we refer to a *kernel* as any small to medium-sized computation consisting primarily of loops. The performance of many HPC applications is heavily dependent on the performance of a few key kernels, which would be the target for our analysis and optimization efforts. Unlike a library, Meliora does not aim to create a repository of ready-to-use functions optimized for particular architectures; rather, it provides a mechanism to *discover* successful optimization of *similar* (but rarely identical) computations. Also, unlike library approaches, there is no specific limitation to the types of computation that can be considered.

The Meliora framework consists of two major components: front end for data collection and back end for data analysis. Figure 21 shows the overall process of performing the front-end analysis to extract the code representation (step 1, Sec. 4.3.1) and the data analysis (steps 2–4, Sec. 4.3.4–4.3.5) in the backend.

4.3.1 The Hybrid Control Flow Graph

The first step in extracting a code representation in Meliora is based on the traditional control-flow graph analysis. A control flow graph (CFG) consists of nodes and edges describing all the possible execution paths of a program. The traditional CFG only contains nodes and edges that can provide limited information, such as the number of basic blocks and their connectivity. We can easily envision two codes with identical CFGs, but vastly different computations within basic blocks. For the purpose of precisely describing the structure and potential runtime behavior of a kernel, we require more information; hence, we introduce the hybrid CFG.

Definition 1 (hybrid Control Flow Graph (hCFG)). *A directed graph denoted as $G = \langle V, E, \delta, \epsilon \rangle$ where vertex V and edge set $E \subseteq V \times V$ stand for basic blocks and directed edges which connect them. In feature sets δ and ϵ, $\delta_i(v_n)$ represents the information attached to node v_n and $\epsilon_j(e_{mn})$ indicates the features attached to the edge from node v_m to node v_n.*

The node and edge attributes are used for learning a representative model in the data analysis phase, which differentiates hCFG from regular CFG. Figure 22 shows the generated hCFG for a matrix-matrix multiplication kernel as Listing 4.1.

Listing 4.1 Matrix-matrix multiplication

```
void multiply (double mat1[][N],
                    double mat2[][N],
                        double res[][N]) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            res[i][j] = 0.0;
            for (k = 0; k < N; k++)
                res[i][j] += mat1[i][k] *
                             mat2[k][j];
        }
    }
}
```

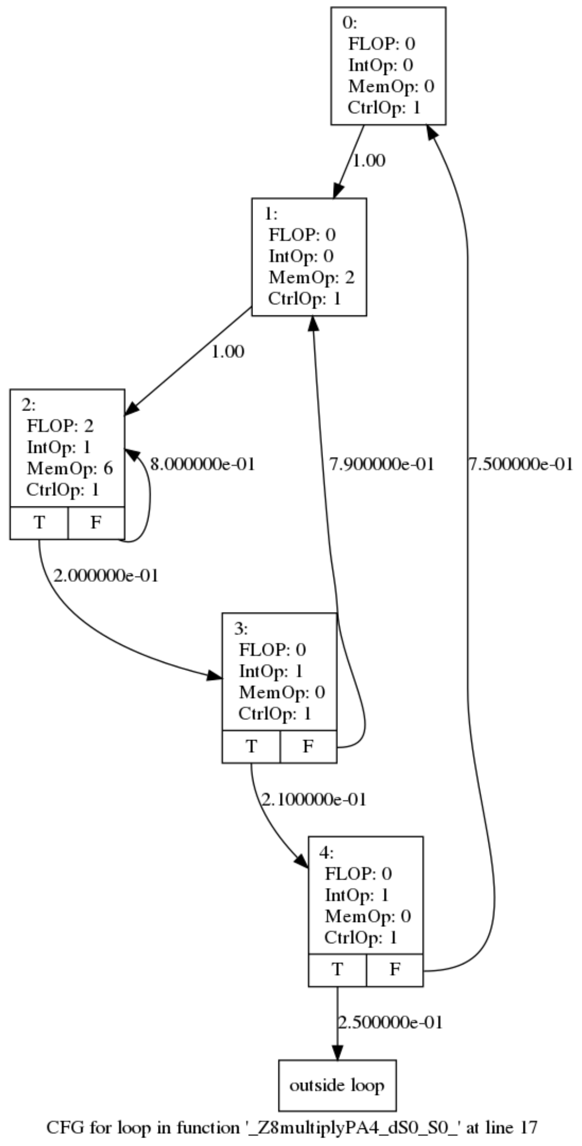


Figure 22. hCFG for the code fragment in List 4.1 (Meliora-generated graph).

The graph structure of the hCFG is the same as that of the regular CFG, in which each vertex represents a basic block including a sequence of operations and each edge indicates the direction of execution flow. In addition, hCFG node attributes include a vector with instruction mix information. Edges are also annotated with attributes described later in more detail. To describe each node, we employ an instruction mix that consists of aggregated instruction counts of four major groups: floating-point, integer, memory access, and control operations. The node attributes categorize every node into different functional clusters by their dominating operation. For example, a floating-point-intensive node is one that contains a large portion of floating-point instructions. Similarly, a node with a majority of memory access instruction would tend to be memory-bound at runtime.

Next, we compute the transition probabilities of each node (by using the method described in [82]) and attach the probability as an edge attribute. Dynamic graph attributes distinguish our hCFG from the attributed CFGs used in security research [44]. They characterize the dynamic behavior of a kernel by lightweight runtime profiling, which offers insightful information about the propagation of values of interest but is not concerned with performance-relevant attributes.
In Table 6 we list the selected node attributes (shown as LLVM IR type) and edge attributes, and we describe the process of extracting these attributes in the following sections.

Table 6. Graph attributes in hCFG

Type	**Attach to**	**Source**
Static	Node	Floating-point: FAdd, FMul, FDiv, FSub
		Integer: Add, Mul, UDiv, SDiv, Sub
		Memory: Store, Load, Fence, GetElementPtr
		Control: Ret, Br, Switch, Resume
		Memory Access estimation
		Loop Depth
Dynamic / Static	Edge	Transition probability: runtime profiling / Statically derived from IR

4.3.2 hCFG Metrics

The metrics added to the hCFG directly affect the performance of the entire Meliora workflow and the accuracy of the generated model reflecting over the prediction results. For the selection of appropriate metrics as model features, we consider three aspects:

1. *Representability*: The metrics should be representative to describe and differentiate a program (or part of a program) which is of importance for feature learning. Moreover, a small number of representative metrics can greatly reduce the costs of model optimization.

2. *Generation complexity*: As a lightweight modeling tool, one goal of Meliora is to provide efficient tuning guidance. Therefore on top of the above requirement, we also consider the complexity of metrics generation in terms of time and resource consumption.

3. *Integration compatibility*: Meliora can work with autotuners such as Orio [105, 55] in postmortem mode and also can behave as a standalone compiler. Thus, it requires the availability of all the metrics during compile time so that Meliora can extract the metrics without requiring application execution.

Table 6 lists all the metrics required for building the hCFG and generating models based on the previous criteria. The rest of this section will discuss the metrics in detail.

Instruction mix is the collection of instructions counters which fall into four categories: Floating-point, Integer, Memory, and Control instructions. The instructions represent both high-level application information and the architecture on which the code will run. If we are looking for ways to accurately characterize each basic block, instructions are a natural choice. The Mira framework [91], for example, demonstrated the applicability and precision of the collection of instructions by static analysis.

The type of a basic block is dominated by the majority category of its instructions. Accordingly, the nodes in hCFG exhibit different behaviors; for instance, some nodes are memory-intensive (e.g., a data structure initialization loop), while others are dominated by computation (e.g., matrix-matrix product). Therefore, instead of storing all instructions, we can compress the data attached to each node to capture just the four categories of instructions and thus further optimize the performance of model generation. Besides, by counting instructions, we are able to derive other valuable metrics, for example, the arithmetic intensity of floating-point operations, to help researchers gain a deeper understanding of their code. All the stated reasons make the instruction mix an ideal metric for Meliora to extract and use to generate models.

Loop depth is used as a metric for the more complicated loop scenarios, for example, nested loops. Since the most computation-intensive kernels consist of one or multiple nested loops, loop depth plays a critical role in providing precise information about the relative location of each basic block. Moreover, loop depth well balances between the complexity of retrieval and representability, which makes it a good candidate for hCFG.

The base level of loop depth is decided by the location of the entry block of the loop. In the program analysis performed in a top-down manner, such information is passed down and carried by each basic block and propagated accordingly (increased by one when a new loop header is encountered).

Memory access estimation. In addition to the instruction mix and loop depth, Meliora also generates reuse distance histograms to abstract the pattern of memory access within a basic block. Previous research [37, 13] shows that the reuse distance is an effective representation of data access locality and for further understanding of the behavior of memory hierarchy. This metric can be seen as a supplement to the instruction mix. This increases the dimension of the node attributes in the hCFG by the size of the reuse distance histogram (customizable, with default size 5), but it enables better differentiation between basic blocks than a simple instruction mix attribute.

The generation of the memory access estimation, however, is challenging for static tools. Most previous research on reuse distance analysis relies on the program instrumentation techniques as they are able to obtain the dynamic information from executions, such as memory references and instruction tracing, etc. Although such data may not be available in compile time, we can offer the reasonably estimated metrics derived from static program analysis, for instance, the range, upper (lower) bounds. In Meliora, the memory access pattern is represented by a histogram vector whose number and size of bins are fully customizable by users. We describe the details of the implementation in the next section.

Transition probabilit. Besides vertex-attached metrics, the transition probability is used as the edge weight, which indicates the probability of the execution flow from one basic block to another [82]. In real environments, it is not usually possible for a program to traverse the execution paths equally, especially for multiple or deeply-nested branches. For example, in the code shown in List 4.1

certain basic blocks are executed more often than others depending on the loop static control part (SCoP). Consequently, it would affect the accuracy of the model if all execution paths were treated as equally likely. Therefore transition probability can be utilized to enhance the structural representability of the hCFG in which the frequent paths carry larger values than the infrequent ones.

In Meliora, the transition probability is the only metric that can be extracted through either static or dynamic analysis. As the unique edge attribute for the graph, it is of great importance; thus, the data obtained from runtime profiling can describe the structure with emphasis on precision. However, in many cases generating the transition probability at compile time is sufficiently accurate and enables the entire Meliora workflow to run in pure static mode, which reduces the time for both creating and using the models.

4.3.3 Metric Extraction

As we mentioned in Sec. 4.3.1, all graph features can be computed statically, and some (transition probabilities) can be optionally computed dynamically. In this section, we describe the static and dynamic techniques used in Meliora for obtaining the features and building the hCFG.

Loops are common and significant code structures in high-performance computing applications. Most computationally intensive code is expressed in loops, and they account for the majority of application performance hotspots. In the paper by Bastoul et al. [9], the authors surveyed several high-performance applications. Their work shows that the average loop coverage, calculated as the ratio of in-loop statements to the total number of statements, ranges from 77% to 100% with an average above 80% for the surveyed applications.

The critical role and the characteristics of a loop make it the ideal target for code optimization. For instance, an optimizer (human, compiler, or autotuner) can apply several techniques such as loop fusion, loop unrolling, and loop tilting to reduce the cache miss rate by the enhancement of locality of the memory system in order to improve the overall performance of the generated code. Hence, Meliora focuses on the loops within kernels for extracting metrics and metrics and assisting in discovering code optimizations.

Analysis granularity. Kernels frequently contain more than one same-level loop. Each of these loops may have unique characteristics that would require a different approach to optimization. In other words, a single optimization strategy for the whole multi-loop kernel is unlikely to yield the best results. Hence, Meliora supports model generation based on metrics obtained from each top-level loop (or nested loops) so that later Meliora can generate the model and guide the optimization matching the same granularity in new kernels. In addition to the loop-basis analysis, we also provide users with the option for performing the static analysis in a coarser-grained manner, for example, running on the function level with all loops combined. The goal of offering a high-level perspective is to further help the users to gain an understanding of the functionality of the kernels as a whole. If a kernel-level match is found, for example, that can help optimize larger portions of the code at once, thereby saving time and effort.

Static analysis for metric extraction. In the design of the analyzer, we focus on efficiency and usability. Therefore, users are able to obtain all the hCFG features during compile time, while dynamic profiling is an available option to choose in case more precise data is needed. In the pure static mode, the static analyzer is utilized for collecting both the node and edge features. We build

Algorithm 1: Memory access estimation generation.

```
Data: hCFG $G$
Result: vector $v$ as the histogram
current byte tracker $bt = 0$;
while $G$ has basic block $BB_k$ not visited do
    initialize local storage $lst$;
    while $BB_k$ has instruction $I_m$ not visited do
        if $I_m$ is store/load instruction then
            update $bt$;
            obtain operand $opd$ from $I_m$;
            if $opd$ is GEPOperator then
                parse $opd$;
            end
            update $lst$;
        end
    end
    update $v$;
end
```

the static analyzer on top of the LLVM intermediate representation (IR), which is the bridge between lexer, parser (frontend), and code generation (backend). In addition to better usability, when performing analysis on the IR level, it is possible for us to isolate the design of the processing logic from the types of source code to make Meliora a language-independent framework. Furthermore, it also reduces the complexity of the analysis compared to working directly on the binary code. Moreover, LLVM provides a large number of handy functions for developers to leverage. Technically, Meliora is expected to work compatibly with any programming language as long as it can be transformed into the LLVM IR. In the rest of this section, we discuss the details of the implementation of Meliora for metric extraction.

Independent of the types of the target metrics, we must inevitably traverse the abstract syntax tree (AST) or similar wrapped structure in LLVM IR one or more times while keeping track of several values in order to summarize the corresponding metrics correctly. However, we can minimize the repeated process by adjusting the entry point of the analysis and also by aligning it with the analysis granularity. Specifically, we consider loops, especially nested loops, as a whole graph so that we can flatten them from the outermost loop. After locating the top-level loop, we treat each basic block equally independent of its type (e.g., loop SCoP) and traverse once to collect necessary data from basic blocks. Each basic block is examined to retrieve and categorize the parsed instructions. For coarser-grained results, we want to collect kernel-level information, which might comprise several loops at the same level. To address this problem, we first identify and locate the first and last loops then generate a fake loop body to enclose them. After that, we can reuse the same method to process the fake loop and generate a kernel graph. This approach also allows finer control over granularity within each kernel that falls between kernel-level and single-loop.

The edge features can be extracted either statically or dynamically. The dynamic method requires program instrumentation. Then the instrumented code is compiled and run to generate the hCFG transition probability edge attributes. Before Meliora comes into play, the profiled data must be merged back with the source code to create the LLVM bitcode. By contrast, in the pure static approach, we provide an LLVM component for obtaining the edge data statically, which uses heuristics to compute the edge probability based on the weights produced by the DAG analysis. We note that both the static and dynamic approaches occur before the loop traversal pass, and the edge data are collected at the same time as node attributes. However, due to the extra steps of instrumentation and execution, the time consumption for the dynamic approach is substantially higher than that of the static method. In section 4.4 we compare the two approaches in terms of time cost and accuracy.

Describing the memory access pattern without profiling data is never a trivial task. Moreover, the enforcement of the static single-assignment (SSA) form in LLVM IR complicates the implementation at the symbolic level. To address the challenges, Meliora employs the symbols extracted from the IR to estimate the bounds of the reuse distance in bytes. This is to say that we might not be able to obtain the precise memory references of an array, yet we can deduce the maximum and the minimum number of access of the same array by appropriate assumptions to compute reuse distance bounds. In the implementation, we iterate the instructions and three LLVM instruction types involve: *StoreInst*, *LoadInst* and *GetElementPtrInst*. As shown in algorithm 6, we start by testing the instruction type, and then for the qualified instructions, the operand of the instruction is visited. Again the operand is tested because the instruction might manipulate the scalar values. Generally, if the operand is an LLVM *GEPOperator*, the algorithm considers the instruction as a memory access operation. Subsequently, the instruction operand is parsed to retrieve information about the array. We might need to recursively parse the operand if the target is a multi-dimensional array. The memory access estimation is generated at the same time as other metrics; no additional graph traversal is required.

4.3.4 Graph Representation Learning

Graph representation learning is at the heart of the Meliora workflow. The framework relies on machine learning techniques to train the model for unseen graph prediction in order to assist the selection of the tuning parameters for code optimization. In addition to the model, it converts the raw hCFGs generated in the front end into a vector while preserving significant graph properties. The embedded form reduces the costs of storage and computation on the original graphs, which is crucial for scaling up this approach to a large number of computational patterns. A number of graph embedding options exist, as briefly discussed in Sec. 4.2. To choose the method most suitable to our needs, we consider both the current demand and extensibility potential. We have four requirements for any potential approach: 1) It must work for arbitrary graphs; 2) It can process auxiliary data besides the topological information of the graph; 3) It can handle both directed and undirected graphs; and 4) It can handle both vertex and edge attributes with discrete and continuous values to generate a whole-graph embedding. Relatively few approaches fulfill these requirements, leading us to the choice described below.

We build the component for graph representation learning on top of PSCN [104] proposed by Niepert et al. This approach is based on convolutional neural networks (CNNs) [75, 74], aiming to learn the arbitrary graph with node and edge attributes for prediction. Because our ultimate goal is to compare and match large numbers of computational kernels, we need an approach that is both *accurate* and computationally *efficient*. If we were to use the graph representation directly, e.g., to build a tree-based classification model, we would have to employ an expensive graph comparison operation, such as graph isomorphism. This would limit both the size and the number of codes that can be considered. Hence, we choose to reduce the hCFG to a vector representation, a task for which CNNs are one of the most suitable approaches. The vector representation also offers the advantage of easy comparisons using a number of different distance techniques, including cosine or Euclidean distances. In Fig. 24, for example, we show the cosine distances between the vector embeddings of test dataset loops and the embeddings of the loops in the training kernels.

The whole procedure consists of three steps. Briefly, *Node sequence selection* is to construct the sequence of nodes and create the corresponding receptive fields. Followed by *Neighborhood assembly* it assembles a local neighborhood for the nodes in the sequence as the candidate for the receptive field. The third step *Graph normalization* is to normalize the neighborhood graph assembled in the previous step by imposing an order on the graph in order to create the vector representation. In order to minimize the overfitting of model training, dropout regularization [124] is applied on the hidden layer of the CNN. The dropout rate is set to 50%. The procedure also trains a model utilized by Meliora to predict unseen graphs.

4.3.5 Using the Model

After the model generation, users can apply Meliora to key loops in their code to locate the best match for an arbitrary graph with the kernels in the model. To achieve that, first, Meliora compiles the source code in any language supported by LLVM into bitcode and then performs the static analysis described in Sec. 4.3.3 on the bitcode to collect the hCFGs representing the loops of the target kernel. One graph is created for each loop in the loop-level mode; otherwise, a single graph for the entire kernel is generated. Subsequently, Meliora uses the model to make predictions consisting of the coefficient vector containing the probabilities of the given graph being similar to kernels used in training. We then choose from this vector the loop or kernel with the largest coefficient, i.e., the best match. At present, this is where Meliora stops, but in the future, we plan to integrate it more closely into the autotuning process. We note that this approach can be used for different types of optimization workflows, not just autotuning. As our training data grows, we anticipate that we would incorporate both manually optimized and autotuned code versions, so depending on the specific match, the user may embark on a manual optimization, code replacement (if a better implementation of the same functionality was located), or enlist the help of an autotuner. We include some specific examples of a possible autotuning integration workflow in the evaluation Section 4.4.3.

4.4 Evaluation

In this section, we evaluate the accuracy of the model and the performance of the process of model generation. The datasets we used to build the model are from the SPAPT [6] benchmark.

4.4.1 Experiment Environment

The machine we used to build the model and validate it is an Intel® Xeon® E5-2699v3 2.30GHz with two 18-core Haswell CPUs and 256GB of memory.

4.4.2 Dataset Generation

It is not easy to collect a sufficient amount of code as the input for training. Hence we create our dataset from scratch for the selected kernels. As we mentioned, Meliora is capable of running in a postmortem mode to work in conjunction with the Orio autotuning framework [105, 55]. In that scenario, Meliora serves as a post-processor invoked by the autotuner to perform the static analysis on the various versions of the tuned code generated by Orio. No

Listing 4.2 Annotated AXPY-4 kernel

```
  // Tuning specification
  /*@ begin PerfTuning (
    // ...
    def performance_counter {
     arg method = 'basic timer';
     arg repetitions = 60;
    }
    def performance_params {
     param UF[] = range(1,33);
    }
    //  ...omitted ...
    def search {
     arg algorithm = 'Randomsearch';
    }
 ) @*/
  /*@ begin Loop (
    transform Unroll(ufactor=UF)
    for (i=0; i<=N-1; i++)
        y[i] = y[i] + a1*x1[i] + a2*x2[i]
                + a3*x3[i] + a4*x4[i];
  ) @*/

  // original C code kernel
  for (i=0; i<=N-1; i++)
    y[i] = y[i] + a1*x1[i] + a2*x2[i]
                + a3*x3[i] + a4*x4[i];

  /*@ end @*/
  /*@ end @*/
```

modifications to Orio were necessary; we believe integration with other autotuners can be accomplished similarly.

The Orio autotuning framework parses the annotations in the source code and generates different versions of the optimized code. Next, Orio empirically evaluates all the generated versions to select the one with the best performance for the production environment. Listing 4.2 shows a portion of the annotated AXPY-4 kernel. In the example, the *performance_params* defines only one performance parameter, unroll factor (UF), ranging from 1 to 33. Orio employs a search strategy (as specified in the tuning spec comment) to determine the optimal unrolling factor for this loop on the platform of interest by executing several different code versions corresponding to different unroll factors. *input_params* defines two different problem sizes that the entire tuning process will repeat for each value in order to search for the best performance for each size.

Table 7. Selected kernels for dataset generation

Kernel	**Operation**
Elementary linear algebra kernels	
GEMVER	scalar, vector, and matrix multiplication
MVT	matrix vector product and transpose
Stencil code kernel	
Stencil3d	3D stencil computation
Linear solver kernel	
BiCG	subkernel of BiCGStab linear solver
Elementary statistical computing kernel	
COV	covariance computation

The dataset we used for training is a portion of the SPAPT benchmark suite [6, 123], which contains four types of selected kernels. Table 7 shows the kernel name and major operations grouped by the categories.

1. *Elementary linear algebra kernels* focus on the mathematical computations on scalars, vectors, and matrices.

2. The *Stencil code kernel* is frequently used for solving partial differential equations. They follow a particular pattern to access and modify the array elements.

3. The *Linear solver kernels* are used in solving systems of linear equations. For instance, the **BiCGStab** linear solver kernel decomposes a matrix into a product of lower and upper triangular matrices.

4. *Elementary statistical computing kernels* refer to the code that helps find the statistical relationship among random variables. The dataset used in the evaluation is listed as the following. Each of them is split into two groups for training and validation, respectively.

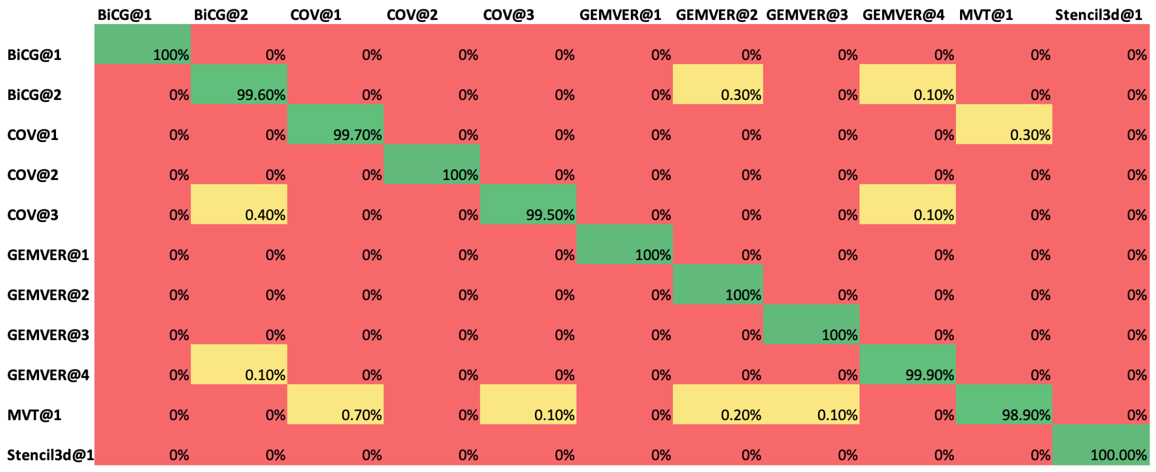
The dataset containing loop versions is statically generated and is divided into two subsets: 38,000 graphs for modeling and 11,000 graphs for validation. The test graphs are versions of the same kernels and can be used to confirm that the matching is able to identify such known similar codes correctly (Figures 23 shows the self-validation loop-level granularity; in addition, we validated with codes not used in training 24, as discussed later).

4.4.3 Results

In this section, we discuss the accuracy of Meliora in identifying the kernels, which we split into two parts: (i) the accuracy for recognizing different versions of kernels used in training and (ii) the effectiveness of using Meliora to optimize completely arbitrary (unseen) code.

1) Model Validation. As we described in Sec. 4.4.2, we split a subset of the kernels (Table 7) in the SPAPT benchmark suite into two groups, using the majority for the model training, while the rest of the code versions are reserved for model validation. In this scenario, we are able to test the accuracy of the model for recognizing different versions of the same kernels used in training.

Figure 23 shows the self-validation results on each loop from the dataset used for training. By self-validation, we mean selecting a loop from a transformed (by Orio) kernel version that was not used in training and computing its match; for example, we expect that most versions of GEMVER would be matched with other versions of GEMVER. This is not a completely trivial validation since many of the transformations impact the hCFG and, to a lesser extent, the instruction mix. The labels on the X-axis and Y-axis are the same as the kernel names, where the X-axis represents all the available classes in the training set corresponding to each of the selected kernels, and the Y-axis indicates the percentage of the graphs in the validation set predicted as the existing kernels. The color of the tiles shows the value of the percentage; the redder the tile is, the closer the value to 0, and similarly, greener means the value is closer to 100.



	BiCG@1	BiCG@2	COV@1	COV@2	COV@3	GEMVER@1	GEMVER@2	GEMVER@3	GEMVER@4	MVT@1	Stencil3d@1
BiCG@1	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
BiCG@2	0%	99.60%	0%	0%	0%	0%	0.30%	0%	0.10%	0%	0%
COV@1	0%	0%	99.70%	0%	0%	0%	0%	0%	0%	0.30%	0%
COV@2	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%
COV@3	0%	0.40%	0%	0%	99.50%	0%	0%	0%	0.10%	0%	0%
GEMVER@1	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%
GEMVER@2	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%
GEMVER@3	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
GEMVER@4	0%	0.10%	0%	0%	0%	0%	0%	0%	99.90%	0%	0%
MVT@1	0%	0%	0.70%	0%	0.10%	0%	0.20%	0.10%	0%	98.90%	0%
Stencil3d@1	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100.00%

Figure 23. Model self-validation at loop-level granularity. BiCG@1 indicates the first loop extracted from BiCG kernel in a top-down manner.

From Figure 23 we can see that most of the cells on the diagonal are colored dark green, which shows that the majority of the graphs are correctly predicted. In particular, the validation results are all above 98% ranging from 98.90% to 100%. On the other hand, the percentage of misclassified graphs is less than 1.1%. Stencil3D shows the best accuracy with all the graphs correctly labeled as itself; Graphs from MVT are the most misclassified results in the 1.1% of the graphs classified as each of the other training classes. In practice, for a given graph, we perform prediction and then choose the most likely matching result based on the prediction vector, which includes the probability of a likely matching against each class in the model. In other words, we always choose the kernel with the highest value.

2) Evaluation of New Kernels. Most real-world use cases of Meliora would utilize the model to find the best match between unknown, arbitrary kernels and those in the model so that we can apply existing optimization knowledge to avoid the time-consuming search on the large variant spaces of performance

Table 8. Meliora's matches for a set of new codes' loops. The `adi@132` notation indicates the loop starting at line 132 of the `adi` code.

New code Name@LoopLoc.	**Matched kernel**	**Coeff.** (<= 1)
adi@132	gemver	0.99
adi@137	gemver	0.99
correlation@166	covariance	0.99
correlation@172	mvt	0.99
correlation@180	mvt	0.85
correlation@185	covariance	1.00
fdtd@152	bicgkernel	0.99
fdtd@154	mvt	0.88
fdtd@157	mvt	0.88
fdtd@160	mvt	0.83
jacobi@76	gemver	0.99
tensor@130	stencil3d	1.00
trmm@124	covariance	0.99
trmm@130	stencil3d	1.00
dgemv@255	bicgkernel	0.83
dgemv@258	gemver	1.00
dgemv@260	bicgkernel	0.83
dgemv@263	gemver	1.00
dgemv@265	bicgkernel	0.91
dgemv@268	gemver	1.00
dgemv@270	bicgkernel	0.99
dgemv@276	gemver	1.00

optimizations. To demonstrate the application of Meliora to new codes, we present results of using it on a subset of the SPAPT benchmarks that *were not* used at all for building the model.

We performed the empirical evaluation on a slightly different Haswell system than the server used in the model generation (for scaling and cost reasons); the system we used was a cluster of Intel® Xeon® CPU E5-2690 v4 @ 2.60GHz nodes. Each node has two 14-core CPUs and 128GB of memory.

The evaluation procedure consists of the following steps. First, we apply Meliora as described in Sec. 4.3.5 to the subset of SPAPT codes shown in Table 8.

None of these computations were used for training. The codes typically contain several loops with various nesting depths and range in size from tens to hundreds of lines of C code.

Train / Test	bicgkernel@89	bicgkernel@91	covariance@143	covariance@149	gemver@130	gemver@134	gemver@140	gemver@143	mvt@105	stencil3d@140	stencil3d@144
adi@132	0.24	0.96	0.2	0.32	0.28	0.98	0.3	0.9	0.81	0.07	0.04
adi@127	0.24	0.96	0.2	0.32	0.28	0.98	0.3	0.9	0.81	0.07	0.04
correlation@166	0.11	0.15	1	0.9	0.21	0.14	0.27	0.33	0.63	0.59	0.44
correlation@172	0.03	0.43	0.83	0.76	0.17	0.46	0.31	0.61	0.86	0.5	0.36
correlation@180	0.17	0.87	0.45	0.59	0.19	0.86	0.2	0.81	0.95	0.23	0.17
correlation@185	0.07	0.06	0.73	0.59	0.03	0.04	0.03	0.08	0.42	0.88	0.75
fdtd@152	0.98	0.35	0.13	0.26	0.73	0.29	0.35	0.25	0.07	0.11	0.24
fdtd@154	0.2	0.79	0.6	0.73	0.23	0.77	0.25	0.79	0.98	0.27	0.2
fdtd@157	0.2	0.79	0.6	0.73	0.23	0.77	0.25	0.79	0.98	0.27	0.2
fdtd@160	0.15	0.86	0.45	0.55	0.24	0.88	0.31	0.88	0.95	0.2	0.13
jacobi@76	0.34	0.91	0.39	0.51	0.44	0.93	0.45	0.94	0.86	0.12	0.06
tensor@130	0.09	0.04	0.47	0.36	0.04	0.03	0.05	0.04	0.27	0.67	0.81
trmm@124	0.02	0.08	0.74	0.55	0.08	0.06	0.14	0.12	0.45	0.88	0.67
trmm@130	0.1	0.03	0.63	0.5	0.07	0.02	0.08	0.06	0.34	0.97	0.9
dgemv@255	0.39	0.94	0.36	0.53	0.38	0.91	0.31	0.88	0.85	0.12	0.08
dgemv@258	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07
dgemv@260	0.39	0.94	0.36	0.53	0.38	0.91	0.31	0.88	0.85	0.12	0.08
dgemv@263	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07
dgemv@265	0.29	0.8	0.57	0.74	0.25	0.76	0.2	0.74	0.93	0.27	0.21
dgemv@268	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07
dgemv@270	0.28	0.98	0.18	0.32	0.29	0.97	0.26	0.87	0.79	0.06	0.04
dgemv@276	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07
dgemv@278	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07
dgemv@280	0.74	0.36	0.24	0.24	0.99	0.38	0.89	0.44	0.17	0.09	0.07

Figure 24. Similarity between test kernels and training dataset loops computed as the cosine distance between their hCFG embeddings.

Once the model returns a match, we compute the similarity score (cosine similarity) between the loops in the new kernel and those in the matched kernel to further refine the mapping. For example, the first loop in ADI (adi@132) is matched with the GEMVER kernel, which has four loops. The cosine similarity between the embeddings of the adi@132 and the gemver@134 loops is the highest; hence we finalize the match to be adi@132-gemver@134. For small datasets, one could just compute similarities instead of employing the kernel-level model; however, as the number of kernels in the training dataset grows, this approach would not scale.

Next, for each of the target loops (indicated by their line number in the left column of the table), we manually transfer the optimal tuning parameters from the corresponding loop in the *matched kernel* (middle column). Because we used Orio to create our training data set, we also simultaneously produced autotuned versions of these kernels. Given the kernel name, problem sizes, and architecture, we can then easily look up the set of transformations that produced the best version of that kernel for that problem size and architecture. Figure 24 shows the cosine similarity between the vector representations of the training and test datasets. We show similarities between the original code versions for each training and test kernel.

The next step in our evaluation is to manually copy the tuning spec from the autotuned version of the matched kernel into the new code, adjusting variable names as needed. For example, the **correlation** code has four loops, two of which were matched with **covariance**, and two with **mvt**. In the future, we plan to significantly automate this step while still allowing the user to customize the inserted tuning annotations.

Because the matched kernels often have more than one potential loop, the user either must select the loop from which to copy the tuning options or combine all possible parameters. The first choice eliminates empirical autotuning altogether and is often possible through quick inspection of both codes; code similarities are often obvious to a human once presented with such limited choices. In a few cases, it is difficult to choose just one loop; in these situations, parameters from multiple loops can be combined, and the new code can be empirically autotuned. While this does not completely avoid the cost of empirical tuning, it does reduce it dramatically (we give some more details later in this section). Individual loop matching can potentially be fully automated, although it is not clear yet whether the explosion in complexity is warranted to save some fairly light manual effort—all the results in this section were produced in a few hours, including creating the tuning specs for the new codes and performing the autotuning for some of the kernels.

Table 9. Search speedup for the unseen kernels.

Kernel	**Search speedup**
adi	19
correlation	1126
dgemv	2
fdtd	19
jacobi	10
tensor	14
trmm	17

Kernel performance The speedups obtained by modifying the new codes as described above are shown in Fig. 25. The baseline is the original, unoptimized version, compiled with a recent GCC compiler using the -O3 optimization level. All optimization options in the matched kernels can be seen in the SPAPT benchmark repository [6, 123] and include loop unrolling, cache tiling, register tiling, SIMD pragma insertion, OpenMP parallelization, and scalar replacement. While we used an autotuner to enable rapid application of these optimizations, one could also apply them manually, albeit at a dramatically increased effort (the size of the tuned code is typically much larger than the original, especially when combining multiple transformations).

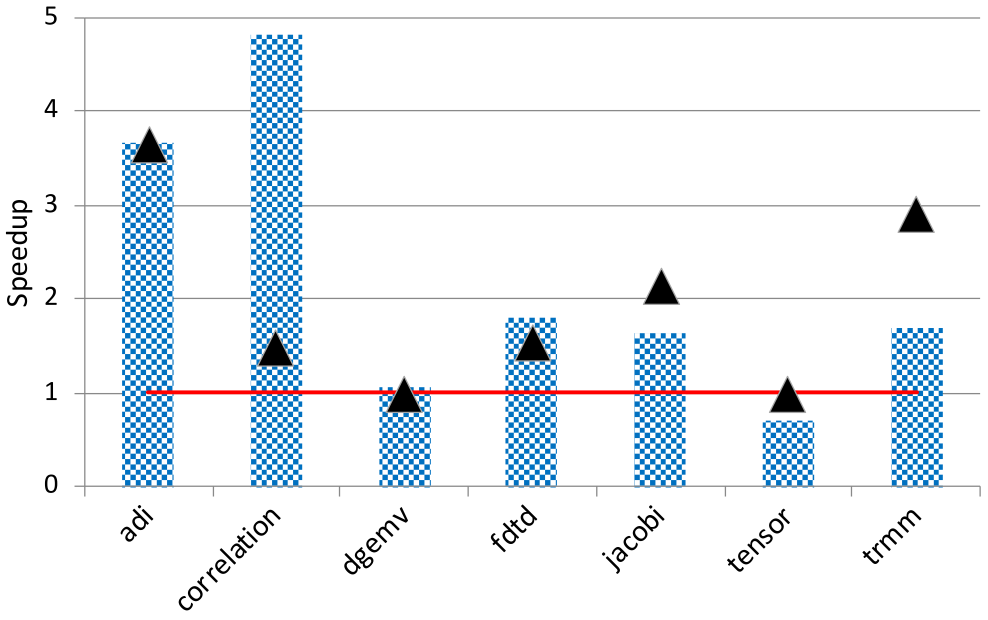


Figure 25. Speedup over the unoptimized (base) versions: bars for Meliora-matched optimizations, black triangles for empirical autotuning results, and a red line for the baseline performance.

We completely eliminated the empirical search *and* produced a better-performing version for some of the new codes (**adi**(1.78x[1]), **correlation**(4.2x), and **trmm**(1.12x)). In addition, we were able to improve performance further by applying limited autotuning for **adi** (3.7x), **dgemv** (1.1x), **fdtd** (1.8x), **jacobi** (1.6x) and **correlation** (4.8x); this required minimal extra effort to modify the

[1]Speedup with respect to the original code version.

parameter space to include the default options. In general, as Table 9 shows, we reduced the search time by 2x to 1,126x for the test kernels, with identical search method configurations.

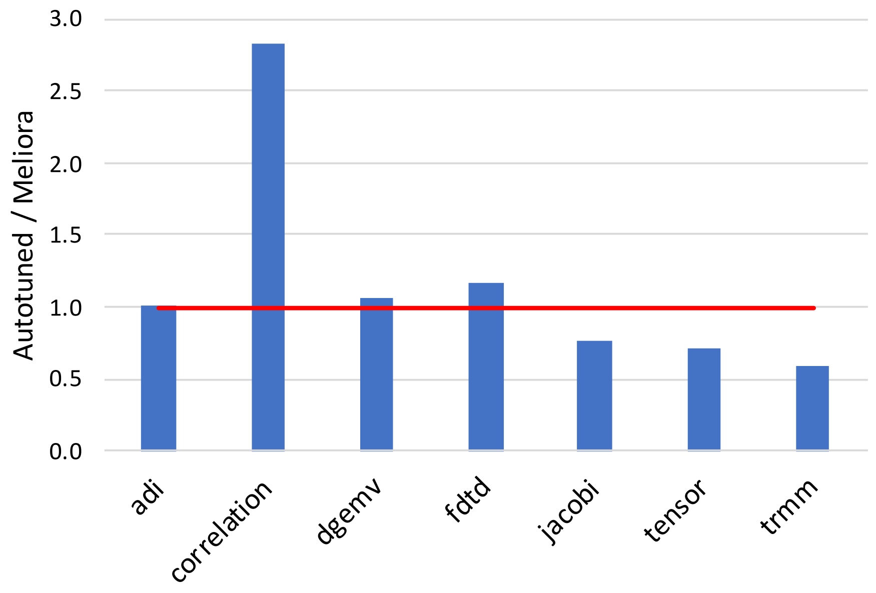


Figure 26. Comparison with empirically tuned performance. Values greater than one indicate that Meliora-based optimizations outperformed the empirical autotuner.

Figure 26 shows a comparison between Meliora-based optimized codes and empirically autotuned versions using a machine-learning-based search strategy capped at 1000 runs (the same as was used to generate the model, although in most cases, fewer than 100 runs were performed per kernel by the search method). Such capping is necessary because the size of the parameter search spaces (ranging from 10^4 to 10^{24} for these codes) is too large to allow exhaustive search. The only code for which the autotuner significantly outperformed the Meliora-based version is **trmm**. For **correlation**, using the tuning specification from the matched kernels significantly outperformed the result from autotuning the original by providing a better starting point for the search, as well as a slightly different set of optimizations.

Autotuning search performance We evaluated performance both on a single set of parameters and with limited autotuning on a small parameter space based on the matched loops. The post-match autotuning also benefits from the use of local optimization methods because we know that the starting point is likely near the optimum.

For the **correlation** benchmark, the Meliora-based optimization through a match with loops from the autotuned **covariance** and **mvt** kernels), we were able to actually outperform previous empirical tuning without any autotuning. The results for **tensor** indicate that this specific benchmark is not optimizable via the kinds of optimizations we attempted (as indicated by the fact that both the Meliora and autotune results are close to the original performance). In part, this is due to the fact that it contains a five-level loop; there is nothing similar among the other kernels in SPAPT. In order to use Meliora for such cases, a model should be trained with a greater number of representative kernels, including tensor contractions.

Performance of metric extraction In addition to the model validation, we time the process of loop-level metric extraction illustrated in Figure 27. The timed process is one component in the workflow of Meliora to generate hCFGs. The plot shows the average time spent on source code compilation (to LLVM IR) and conducting the information retrieval.

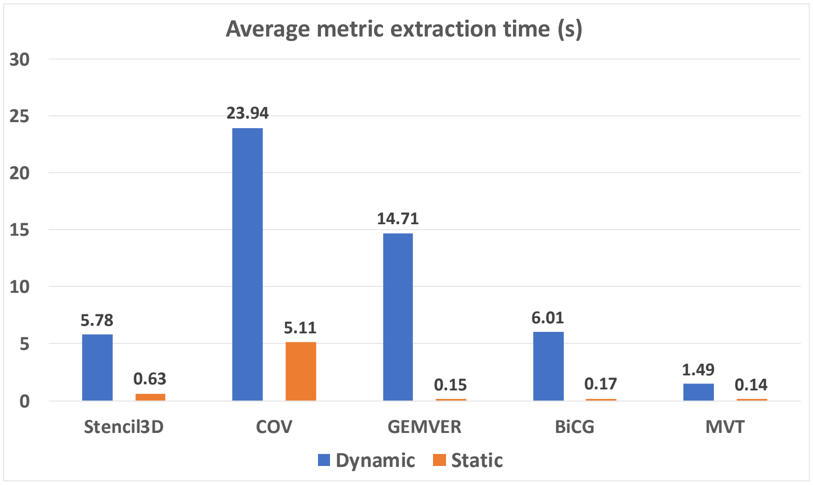


Figure 27. Comparison between the static and dynamic strategies for metric extraction. The X-axis lists the kernels and the Y-axis is the time in seconds.

The metric extraction time varies from kernel to kernel depending on the size of the generated code, which is determined by both the kernel and the input parameters set by Orio annotations. Since the dynamic method demands additional steps to prepare and run the instrumented code, it is not surprising that it takes longer to collect the metrics. Among the kernels, COV (**covariance**) is the most time-consuming, and it needs approximately 5 seconds on average, even for the static processing of the graphs, which is mainly spent on the compilation and I/O operations for the relatively large graphs.

4.5 Related Work

In this section, we briefly survey some works on the application of graph representation learning to performance modeling and model-based performance optimization.

Compiler optimization and autotuning overlap significantly. For instance, iterative compilation [14, 70] exhaustively searches the parameter space for the best-performing combinations. However, compilers apply optimization techniques that can be generally applied to all codes, while autotuners usually focus on specific types of computations and apply more aggressive optimizations. Analytical models derived manually are used at the early stage of compiler optimization. Wagner et al. [135], and Tiwari et al. [130] hand-tuned the analytical model to estimate the execution frequency of code regions and the energy consumption. Nevertheless, creating the model by hand requires expertise and significant effort, and the accuracy and the portability of the model completely depend on human factors.

Machine learning alleviates the complexity of analytical model creation and has been a technique widely used in compiler optimization. Wen et al. [138] and Luk [86] leverages the regression on predicting the application speedup and running time. Classification is also used for locating the optimal optimization parameters [125, 143, 10]. In addition, clustering as an unsupervised learning method is used to select the optimal execution point for program simulation [107]. Compared to the surveyed works, our proposed method concentrates on the intrinsic features of the code as the complementary attributes to the topology information and also emphasizes the automatic feature extraction in order to further reduce the human efforts involved and improve the framework efficiency.

Although our research goals are different, we conduct research motivated by the same ideas, which leverages graph representation learning for similarity detection. Thus it is valuable to mention other such works in this section. For example, Xu et al. [140] and Liu [83] utilize learned the embedded format of the control-flow graph extracted in binary functions to compute the distance in order to measure the binary similarity. They consider only the structure and do not include instruction mix information. Lim et al. [82] define CFG-based similarity metrics for matching GPU kernel computations that also include instruction mix information; that approach is limited to NVIDIA codes, but integration with our CPU approach is a feasible future direction.

4.6 Summary

In this chapter, we introduced Meliora, a framework for extracting code representations that can be used to find potential optimizations for new codes more easily and eventually automatically.

By defining hCFG-based program representations and using them to match computations with previously-optimized kernels, we provide an affirmative answer to our second research question (*"Can we use compiler-generated intermediate representations to create program embeddings that allow accurate matching of loop-based computations?"*). Moreover, we demonstrated how a language- and architecture-independent approach to code matching can be used to eliminate or dramatically reduce the search space of optimizations during autotuning while achieving significant speedup for most kernels in the SPAPT benchmark suite.

In future work, we plan to consider additional code features and automate the use of Meliora in Orio to guide the search without requiring manual interfacing between the two tools.

CHAPTER V

MELIORA-BASED OPTIMIZATION OF CUDA COMPUTATIONS

This chapter is based on the ongoing work by Boyana Norris and myself. In this research, Boyana Norris helped with the generalization of the research idea and direction and provided guidance for organizing the experiment and analysis of the results. I extended the framework with new metrics for modeling the GPU code. I created annotated kernels for generating the dataset, performed model training, and generated matching results. Boyana Norris performed the evaluation of the Meliora-based vs. the Orio-autotuned SPAPT benchmarks.

In this chapter, we apply the Meliora approach described in Chapter 4.3 to the problem of optimizing GPU computations.

5.1 Motivation

In the past decade, we have witnessed a rapid growth in the use of GPUs (Graphics Processing Units) in high-performance computing as they have become ubiquitous in the computing environments available to scientific and engineering applications.

It is the architectural design that differentiates the GPU from the CPU and makes the GPU more suitable for handling parallel code. The reason lies in the fact that the part of the capability of a CPU is diverged to handle the non-computational tasks (e.g., caching). Therefore historically, CPUs have been optimized for processing sequential code. In contrast, the GPU concentrates on computations and has more compute units. To approximately compare the difference in the computation, we use the CPU and the GPU of the similar model year and price range, for example. The 10th generation of Intel Core i9-10900K

(3.7GHz, ten cores, MSRP $429) processor can achieve 592 GFLOPS [64]; While the AMD Radeon RX 5700XT GPU (1.6GHz, 40 compute units, MSRP $399) is able to sustain 9.75 TFLOPS [3]. We can see that although each computational unit in a GPU has less computing capability, the overall performance for the GPU is roughly 16x that of the CPU. Hence, for both the performance and the budget-wise, the GPU is more suitable for parallel code.

Along with the increasing efforts for transiting to GPU-based computing, the demands for application optimization and tuning in order to accommodate the architecture rise dramatically. For sparse linear algebra, researchers focus on the presentations of the sparse matrices to optimize the code generation. Bell et al. [12] mix the use of ELL (ELLPACK/ITPACK [48]) and COO (coordinate) format. The ELL format is used to store most of the matrix elements while the rest of the elements are stored in COO format to reduce the number of zeros in ELL in the scenario that the number of non-zero elements varies significantly for each row. Monakov et al. [97] divide the matrix into smaller slices stored separately in ELL format. This method can greatly reduce the number of zeros in the storage with row reordering. Similarly, Choi et al. [32] modify the ELL format to store the smaller dense blocks of the matrix. Besides the modification of ELL format, both [97] and [32] utilize autotuning to find the optimal parameters for the data structures.

5.2 Approach

In this section, we describe the design and implementation in detail of how we generate and apply the graph-based model to improve the GPU code optimization and generation. Our unified performance optimization framework comprises three major parts: the component for guiding CPU code optimization, the component for assisting GPU code performance optimization, and the autotuner for generating the code for target platforms. Meliora is used for generating code representations in both our earlier work presented in Chapter IV and the work described in this chapter. The goal of the work described in this chapter is to extend our graph-based machine-learning model to greatly reduce the search space during the optimization of *GPU codes* by identifying and reusing prior GPU optimization knowledge.

With this work, we target small to medium-sized (10s-100s of lines) computation-intensive kernels, which usually are the fundamental parts of the HPC applications, and directly decides the overall performance. The output of this GPU component is not the optimized code; instead, it generates the best-matched optimization parameters from the previously optimized code to guide Orio to narrow down the search space exponentially to generate the optimized code. The output of this GPU component is not the optimized code; instead, it generates the best-matched optimization parameters from the previously optimized code to guide Orio to narrow down the search space exponentially to generate the optimized code. When used in the unified model, the GPU component is able to utilize the model trained with Meliora to skip the training process. However, it still requires to analyze the target GPU code for extracting related metrics to use the model for identification. When it runs in standalone mode, the workflow of the GPU component is similar to Meliora. It runs the front-end analyzer for inspecting the target code and obtain necessary metrics and then constructs the graph model using convolution neural networks.

5.2.1 Data Dependency Metrics

In Chapter IV, we introduced several metrics used for representing computational kernels, which are extracted from source code statically, including instruction mix, loop depth, memory access estimation, and transition probability. To capture additional features that reflect available parallelism in the code, we introduce additional dependency metrics for memory accesses. These metrics are considered as supplements to the memory access estimation for the in-loop data dependencies. Having these metrics is essential to provide more accurate matching results for optimizing parallel computations, such as GPU codes.

In this work, we extend Meliora's features with counts for Read-After-Read (RAR), Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies. Obtaining these counters statically is a non-trivial task given that our static analyzer works on the LLVM IR level, which limits the ability to access some data, for example, symbol information. Hence, the analyzer is configured with the preset rules for processing the dependency metrics efficiently. The rules for collecting the dependency metrics include the following.

1. To limit the scope of the analysis, we only consider the statements in the innermost loop when encountering a multi-level nested loop.

2. We count each pair of the read and the write operations as one by checking the load and store instructions. So we skip counting any single remaining read or write operations; and

3. For the load and store instructions, we check whether it accesses the array variables and the indexes of the array are loop-dependent.

In the process of the dependency metric extraction, the analyzer runs in a top-down manner for two reasons. First, it is required for the static analyzer to parse the information at a higher level of the AST and then pass it down to the innermost loops to help with the identification of the dependency metrics. For instance, the loop index variables are parsed at each level of the nested loop and passed down to the innermost loop so that it can determine whether read or write operations to an array should be handled; Second, it aligns with the behaviors for collecting other metrics in order to minimize the number of traverse through the AST. After the target loops are processed, the static analyzer attaches the data to the hCFG. It is worth noting that we have to insert zeros to the attribute vector attached to each node to keep the consistency of the representations for the vertices because only the basic blocks that belong to the innermost loops are analyzed for generating the metrics.

5.2.2 Delivering Loop-to-Loop Matching

As we mentioned in our description of Meliora in Chapter IV, the framework is capable of performing analysis at different granularities, not only for CPU codes but also for GPU codes. It can process and generate the metrics at the function (or kernel) level. This means that the static analyzer will traverse the computation starting from the outermost loop and will treat every structure inside as normal hCFG nodes. The benefit of this coarse-grained process is that it provides a high-level view for understanding the behaviors of the entire kernel, and the metrics obtained at the kernel level can be used to match larger-sized code segments. For example, the `gemver` kernel contains several loops that often occur in a similar sequence in algorithms, such as iterative Krylov methods for solving linear systems.

Being able to match them as a single unit enables the simultaneous optimization of these multiple loops.

Listing 5.1 Bicgkernel from SPAPT benchmark

```
for (i = 0; i <= ny-1; i++)
  s[i] = 0;

for (i = 0; i <= nx-1; i++) {
  q[i] = 0;
  for (j = 0; j <= ny-1; j++) {
    s[j] = s[j] + r[i]*A[i*ny+j];
    q[i] = q[i] + A[i*ny+j]*p[j];
  }
}
```

However, such coarse-grained kernel-level analysis is not always the best option for all applications. When a kernel contains several loops at the same level, one could miss optimization opportunities in certain inner loops or the ability to apply different parameters to tune the inner loops separately from others in the kernel. Similarly, for the kernel with multiple sub-loops, it is not necessary to optimize each loop. For example, Listing 5.1 shows the `bicgkernel` from the SPAPT benchmark [123], which is a sub-kernel of the BiCGStab linear solver. We can see that the first loop in the kernel is used for the initialization of the array preparing for the next step. It is not worth the efforts to optimize the first loop because it has no potential to improve the performance. Instead, we should focus on the optimization of the second nested loop to achieve better overall performance for the `bicgkernel`. Therefore, considering the scenarios that the kernel-level analysis is not able to provide the flexibility for users to select the tuning target, we offer the loop-level analysis and deliver the loop-to-loop results. To be specific, in the code preparation stage, users can specify which loops are of interest to investigate with the annotations. Next, during the metric generation stage, the static analyzer parses the code into AST and then traverses top-down to obtain the necessary data and graph topology for training the model. After that, data is preprocessed and reformatted to accommodate the input requirement of the graph neural networks. Accordingly, users are required to annotate the code sections in the new kernels (unseen ones) as the input for the analyzer to locate and process. After modeling training completes, the model is able to provide the optimal tuning parameters for the best matching code structures against the input for guiding the code generation.

5.2.3 Static Analysis for GPUs

The analyzer runs on the LLVM IR level to provide users a language-independent capability, which theoretically enables our method to be applicable to all the languages that the LLVM front end supports. The static analysis can be performed standalone and in conjunction with Orio. We run static analysis on the unseen kernel for generating the input for the model to match or using as new training data; on the other hand, the static analyzer runs with Orio to perform on the Orio-generated code variants to automatically generate the large dataset for the model construction.

From the viewpoint of the static analyzer, the source code used for the model training is the same. For example, in Listing 4.1, it shows the original kernel for matrix multiplication, which contains a three-level nested loop and a statement for array accesses. For obtaining the code metrics for modeling or matching, the listed code contains sufficient information for performing the static

Listing 5.2 Part of the CPU annotations for matrix multiplication

```
/*@ begin Loop(
  transform Composite(
    tile = [('i',T1_I,'ii'),('j',T1_J,'jj'),('k',T1_K,'kk'),
            (('ii','i'),T1_Ia,'iii'),(('jj','j'),T1_Ja,'jjj'),
            (('kk','k'),T1_Ka,'kkk')],
    unrolljam = (['i','j','k'],[U_I,U_J,U_K]),
    scalarreplace = (SCREP, 'double', 'scv_'),
    vector = (VEC, ['ivdep','vector always']),
    openmp = (OMP, 'omp parallel for private(iii,jjj,kkk,ii,jj,kk,i,j,k)')
  )
  for(i=0; i<=M-1; i++)
    for(j=0; j<=N-1; j++)
      for(k=0; k<=K-1; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
) @*/
```

analysis. Therefore, we can use the same code for constructing the model to guide both CPU and GPU code performance optimization. The benefit of this design lies in the fact that we can significantly reuse the components in our implementation, and also it reduces the tedious and time-consuming work for the users to prepare the inputs. Besides, by unifying the input, we are able to utilize the same graph across different platforms. It is worth noting that to accommodate this unified optimization workflow, the metrics generated by the static analyzer for model training are general and language-independent, which should be available in both CPU and GPU codes. In this section, in addition to the hCFG approach presented in Chapter IV, we also describe the metrics we use to target the representation of the parallelism-relevant features. All the metrics are available to the static analysis tools described earlier.

Listing 5.3 Part of the GPU annotations for matrix multiplication

```
 /*@ begin Loop(
  transform CUDA(threadCount=thread_count,
                 blockCount=block_count,
                 preferL1Size=preferred_L1_cache,
                 unrollInner=inner_loop_unroll_fact,
                 streamCount=stream_count)
  for(i=0; i<=M-1; i++)
      for(j=0; j<=N-1; j++)
        for(k=0; k<=K-1; k++)
          C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
    }
) @*/
```

Although there are no extra steps for the users to modify the code in order to apply the model to the unseen kernel, it is necessary to annotate the source code with minimal effort for generating the dataset for training and performing the autotuning with Orio. These annotations used for CPU and GPU code are slightly different[1]. Listing 5.2 and Listing 5.3 shows the part of the annotations used for CPU and GPU code respectively. We can see that the differences are in the performance tuning parameters, such as tile size, unroll factor, OpenMP parallelization for generating optimized CPU code, while in generating CUDA, the tuning parameters include block, thread, and stream counts, and preferred cache sizes. With the exception of the performance tuning parameters, the C implementations of the kernels, which serve as the input for the Meliora static analysis, are the same for CPUs and GPUs.

[1]Efforts are underway to automate the Orio tuning spec generation based on a simpler one-line annotation, eliminating these differences and further minimizing the need for user input.

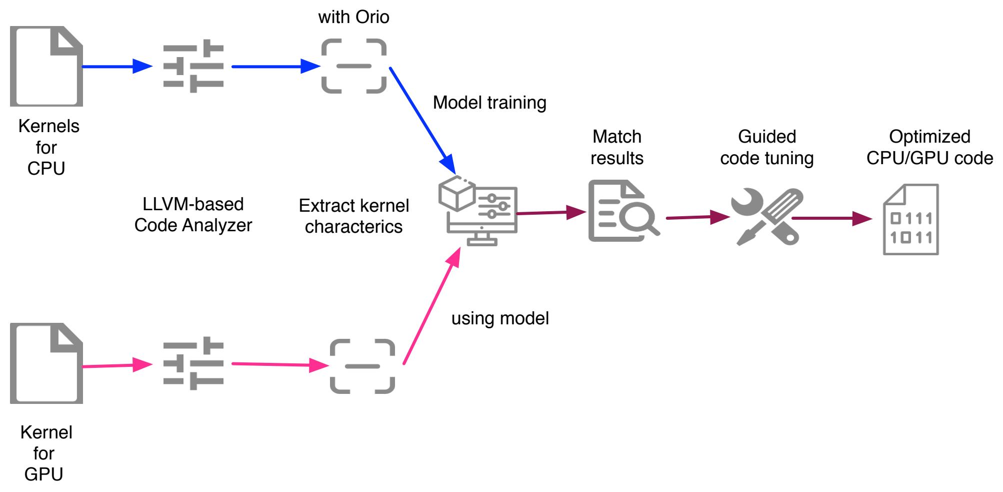


Figure 28. Unified workflow for CPU/GPU code optimization.

5.2.4 Unifying the Workflow for CPU/GPU Code Optimization

As described in Sec. 5.2.3, the use of the same input source code for CPU and GPU autotuning enables us to unify the workflow for CPU and GPU code generation. When matching a new kernel, the framework performs static analysis to obtain the graph representation regardless of the type of the input. Next, the optimal tuning parameters are selected based on the matching results. For each category used for training the model, we can store the tuning parameters for both CPU and GPU code generation and apply them to guiding the code optimization and generation according to the user's choice, either CPU, GPU, or both. The advantage of the unified workflow is that we are able to reuse the trained model for code optimization across platforms, which further reduces the cost of generating the optimized new kernels. Figure 28 shows the unified workflow for CPU and GPU code optimization. The figure demonstrates a typical use case of the unified framework in which we generate a dataset and train the model with Orio [55] for CPU code optimization (highlighted by blue) and then later the process of GPU optimization can directly utilize the model for guiding the code generation. Furthermore, any unmatched input codes are added to the dataset for future retraining of the model to improve its accuracy. It is worth noting that the framework is symmetric, meaning we can reuse or retrain the model from either the CPU or the GPU side.

5.3 Evaluation

In this section, we evaluate the effectiveness of the Meliora-obtained optimization suggestions for GPU code optimizations by using the matching methodology described in Section 4.3 of Chapter IV to SPAPT [6] benchmark.

5.3.1 Experiment Methodology

We used annotated C implementations of the SPAPT kernels, as illustrated in Figure 5.3 for the matrix-matrix product computation. These annotations serve as input to the Orio autotuning framework [105, 55], which is used in two ways for this evaluation.

First, we autotuned the set of *training* benchmarks, e.g., those from which we wish to "learn" how to optimize other codes. They are in the right-most column of Table 8. Because of the large search space dimensions (ranging from $1.23E+05$ to $2.85E+19$ depending on the complexity of the kernel), an exhaustive empirical search is prohibitive. Hence, we used the `Mlsearch` Orio search method in conjunction with the z3 option, which provides several feasible random initial points. `Mlsearch` is based on `sklearn.ensemble.ExtraTreesRegressor`, an extra-tree regressor method, implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

Then, we considered the set of *test* benchmarks (assuming no prior optimization knowledge) and applied the optimization parameters that resulted in the best performance in the corresponding training set benchmark. Note that each loop in the new codes was matched separately, resulting in matching a single new code to one or more training kernels.

We also separately autotuned the test benchmarks to assess how the Meliora-produced optimizations compare with regular autotuning (with the best search method available in Orio).

5.3.2 Results

Mirroring the evaluation approach described in Section 4.4.3, we use the matching results from Table 8 (in Chapter IV) obtained with Meliora to generate and tune CUDA implementations of the test SPAPT kernels.

The baseline performance was obtained for the Orio-generated CUDA code with the parameters in Table 10, which are targeting the NVIDIA A100 SXM4 40 GB GPU used for these experiments. For example, since the A100 can execute 108 blocks simultaneously, we chose 108 as the block count default value.

Table 10. The default parameters in the Orio-generated CUDA implementations of the test SPAPT kernels and their autotuning (AT) ranges in the tuning specifications (using Python syntax).

Parameter	Baseline	AT Values
Thread count	32	range(32,1025,32)
Block count	108	range(108,1081,108)
Inner loop unroll factor	False	range(1, 33)
Cache blocks	False	[False, True]
Preferred L1 cache size	16MB	[16,32,48]
Stream count	1	range(1,33)
Compiler flags	`-O3`	['-O3','-use_fast_math']

Figure 29 shows the speedup resulting from applying the Meliora matching to either eliminate or greatly reduce the empirical search space. For example, the column labeled `correlation` used the matched benchmark's (`covariance`) parameters used for obtaining the best performance when `covariance` was autotuned and achieved 30.3x speedup over the default compiled-optimized CUDA version without performing any empirical autotuning. The columns labeled with kernels with plus signs indicate that very limited (fewer than ten runs) autotuning was performed around the matched kernel's parameters. For example, `correlation+` with limited, targeted autotuning achieved 53.9x speedup over the base version.

We also compared the execution times of the benchmarks optimized with Meliora with those resulting from autotuning with the Mlsearch+z3 Orio search method (the best among approximately ten different available search strategies). Figure 30 shows that, while Meliora is able to deliver good speedups in most cases, there is still room for improvement with traditional autotuning. In three cases, `correlation+`, `tensor`, and `lu`, the Meliora-based optimization actually outperformed the search. Of course, if we could afford to perform an exhaustive

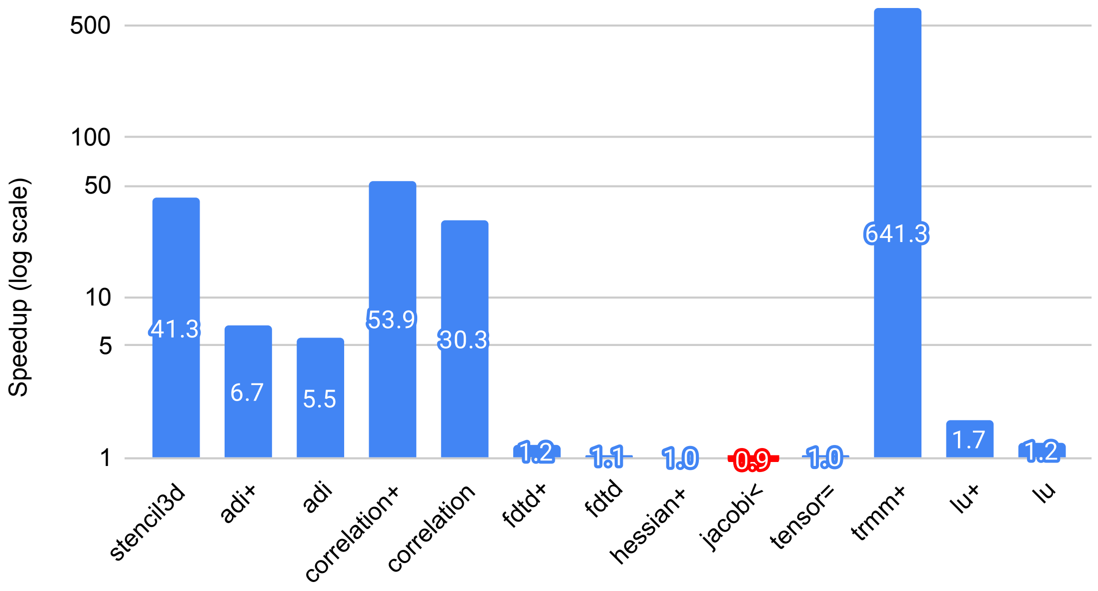


Figure 29. Speedup of Meliora-matched optimizations w.r.t. a baseline version using default parameters and compiler optimizations. In all cases except for `jacobi`, the Meliora versions were at least as fast as the base version, which uses the default optimization parameters in Table 10.

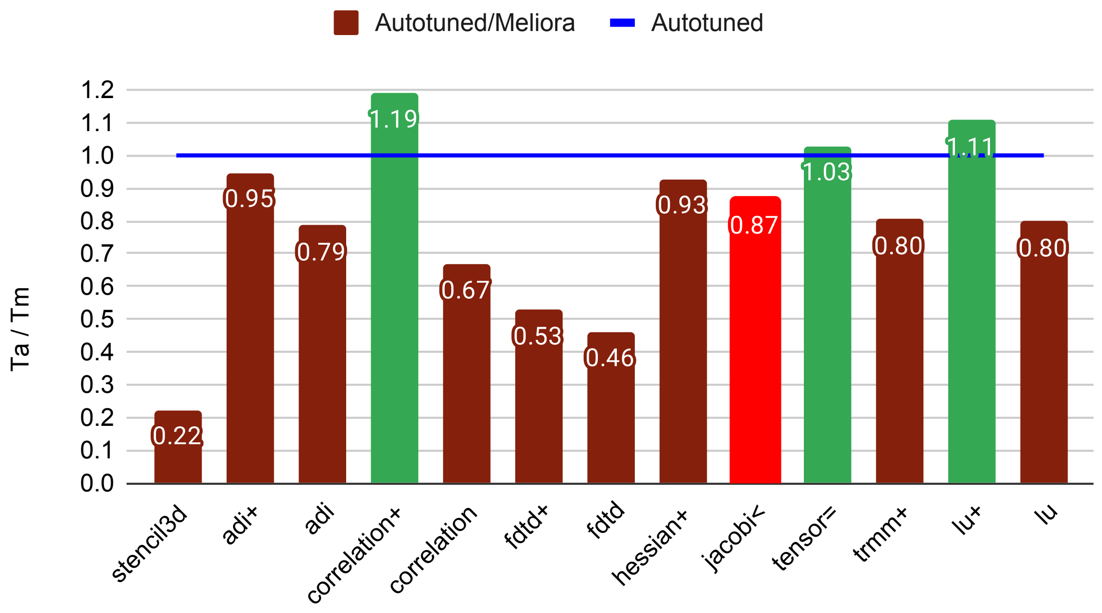


Figure 30. Ratio of autotuning execution time (Ta) to Meliora-based execution time (Tm) of the SPAPT benchmarks. The green bars (values above 1.0) indicate that Meliora-based versions outperformed the traditionally autotuned versions. Values smaller than 1.0 indicate that autotuning can yield additional benefits. In all cases except for `jacobi`, the Meliora versions were faster than the base version.

search, this would not happen, but in practice, exhaustive search is typically infeasible. In half the test benchmark, Meliora delivered performance comparable to that of autotuning.

The final part of the evaluation considers the search speedup resulting from using Meliora, compared with `Mlsearch+z3`-based Orio autotuning. We present the no-tuning results separately in the left plot of Figure 31 since the elimination of the empirical search means that now the cost is 0; however, we did measure the time it took to evaluate the performance of the single version and used the total Orio time for generating, compiling, and testing the single version as the denominator for computing the search speedup. In the right plot of Figure 31, we show the search speedup when we use the Meliora-provided parameters to reduce the search space

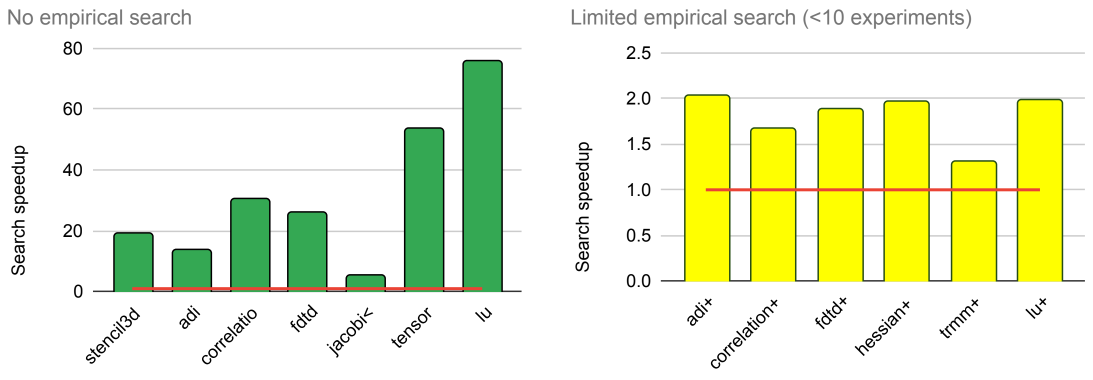


Figure 31. Comparison with empirically tuned performance. Left: Meliora-guided optimizations applied without empirical search (technically the speedup is infinity, but here we show the overall time of evaluating the code version). Right: Restricted empirical autotuning, limiting the search around the Meliora-identified parameters.

and provide a good starting point. In all cases, the search time was reduced from the regular autotuning time, which represents the fastest available search method (in Orio, but also in any autotuner, to our knowledge).

5.4 Summary

In this chapter, we demonstrate the effectiveness of our proposed graph-based program representation, and the derived model can achieve the competitive results for guiding the optimization of GPU codes, which provides a positive answer to RQ3. By the model-guided matching, we greatly reduced the search space for GPU optimization resulting in up to 75x speedup without empirical search for the computation kernels from SPAPT [123] benchmark. In addition to the search speedup, the Meliora-guided optimized CUDA version outperforms all the baseline implementations of SPAPT [123], except for JACOBI, with default optimization parameters. The largest speedup is 643x for the TRMM kernel comparing to the base CUDA version.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

We started this work with the question: What program representations enable the static performance modeling and accurate matching of key computational kernels to support manual or compiler optimizations?

In this thesis, we explored two different approaches to using static program analysis to answer this question and implemented the corresponding techniques in the Mira and Meliora tools. With Mira (Chapter III), we combined static analysis of binaries and source code to produce accurate characterizations of the performance of iterative computations. We based our implementation on the ROSE compiler framework, which provides high-level interfaces to both high-level and binary program information. We demonstrated the accuracy of the performance models by comparing their prediction with hardware counter data obtained with the TAU performance framework. With Meliora (Chapter IV), we created a new control flow graph-based program representations that enable programmers and autotuners to *reuse* optimization knowledge to greatly speed up the optimization process and produce efficient implementations of key numerical computations. We demonstrated that this approach is indeed both language- and architecture-independent by applying an existing model to a new problem—CUDA code optimizations (Chapter V), resulting in completely eliminating the search or reducing it significantly.

6.2 Future Work

While we believe that we have successfully demonstrated that it is indeed possible to devise static analysis techniques to model performance and generate compact program representations, both of which can be used to guide performance optimization, the work presented in this thesis is only the beginning. In future work, we will explore additional features that can improve the accuracy of the hCFG-based matching methodology. We are also investigating using attention and other machine learning techniques to enable the correct matching of computations of larger and more complex codes. Automating the use of Meliora in existing autotuners would enable us to apply this approach more broadly and demonstrate a path forward to adapting it for mainstream compilers.

REFERENCES CITED

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[3] Amd gpu specifications. `https://www.amd.com/en/products/graphics/amd-radeon-rx-5700-xt`.

[4] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[5] R. Bagrodia, E. Deeljman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulations. In *ACM SIGPLAN Notices*, volume 34, pages 151–162. ACM, 1999.

[6] P. Balaprakash, S. M. Wild, and B. Norris. Spapt: Search problems in automatic performance tuning. *Procedia Computer Science*, 9:1959–1968, 2012.

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

[8] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[9] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 209–225. Springer, 2003.

[10] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 307–316. IEEE, 2017.

[11] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[12] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[13] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360. Citeseer, 2001.

[14] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. 1998.

[15] S. Böhm and C. Engelmann. xsim: The extreme-scale simulator. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 280–286. IEEE, 2011.

[16] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, et al. Mambo: a full system simulator for the powerpc architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, 2004.

[17] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[18] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield. New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. *IBM Journal of Research and Development*, 47(5.6):653–670, 2003.

[19] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The international journal of high performance computing applications*, 14(3):189–204, 2000.

[20] D. Bruening and S. Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[21] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[22] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[23] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. Fast multi-parameter performance modeling. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 172–181. IEEE, 2016.

[24] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2013.

[25] L. Carrington, M. Laurenzano, and A. Tiwari. Characterizing large-scale hpc applications through trace extrapolation. *Parallel Processing Letters*, 23(04):1340008, 2013.

[26] L. Carrington, M. A. Laurenzano, and A. Tiwari. Inferring large-scale computation behavior via trace extrapolation. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1667–1674. IEEE, 2013.

[27] L. Carrington, A. Snavely, and N. Wolter. A performance prediction framework for scientific applications. *FUTURE*, 1297:1–11, 2004.

[28] L. Carrington, N. Wolter, and A. Snavely. A framework for application performance prediction to enable scalability understanding. In *Scaling to New Heights Workshop, Pittsburgh*. Citeseer, 2002.

[29] H. Casanova, A. Legrand, and M. Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126–131. IEEE, 2008.

[30] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf. Software design space exploration for exascale combustion co-design. In *International Supercomputing Conference*, pages 196–212. Springer, 2013.

[31] K. M. Chandy and R. Sherman. The conditional-event approach to distributed simulation. Technical report, UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST, 1989.

[32] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM sigplan notices*, 45(5):115–126, 2010.

[33] M. J. Clement and M. J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 886–894. ACM, 1993.

[34] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, volume 28, pages 1–12. ACM, 1993.

[35] P. M. Dickens, P. Heidelberger, and D. M. Nicol. A distributed memory lapse: Parallel simulation of message-passing programs. *ACM SIGSIM Simulation Digest*, 24(1):32–38, 1994.

[36] P. M. Dickens, P. Heidelberger, and D. M. Nicol. Parallelized network simulators for message-passing parallel programs. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995. MASCOTS'95., Proceedings of the Third International Workshop on*, pages 72–76. IEEE, 1995.

[37] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–257, 2003.

[38] J. J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[39] T. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 256–266. IEEE, 1992.

[40] C. Engelmann. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems*, 30:59–65, 2014.

[41] C. Engelmann and T. Naughton. Toward a performance/resilience tool for hardware/software co-design of high-performance computing systems. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 960–969. IEEE, 2013.

[42] T. Estrada, M. Taufer, K. Reed, and D. P. Anderson. Emboinc: An emulator for performance analysis of boinc projects. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[43] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[44] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.

[45] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12(1):1–20, 1989.

[46] K. Fürlinger and M. Gerndt. ompp: A profiling tool for openmp. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008.

[47] M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, University of Passau, 2004.

[48] R. G. Grimes, D. R. Kincaid, and D. M. Young. *ITPACK 2.0 user's guide*. Center for Numerical Analysis, Univ., 1979.

[49] T. Grosser, A. Groesslinger, and C. Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[50] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[51] J. L. Gustafson and R. Todi. Conventional benchmarks as a sample of the performance spectrum. *The Journal of Supercomputing*, 13(3):321–342, 1999.

[52] R. J. Hall. Call path profiling. In *Proceedings of the 14th international conference on Software engineering*, pages 296–306. ACM, 1992.

[53] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[54] F. E. Harrell Jr. Regression modeling strategies. 1995.

[55] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11. IEEE, 2009.

[56] M. A. Heroux, D. W. Doer¡DF¿er, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Sept. 2009.

[57] R. W. Hockney and E. A. Carmona. Comparison of communications on the intel ipsc/860 and touchstone delta. *Parallel Computing*, 18(9):1067–1072, 1992.

[58] T. Hoefler, W. Gropp, W. Kramer, and M. Snir. Performance modeling for systematic performance tuning. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC).*

[59] T. Hoefler, T. Schneider, and A. Lumsdaine. Loggopsim: simulating large-scale applications in the loggops model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, 2010.

[60] T. Hoefler, C. Siebert, and A. Lumsdaine. Group operation assembly language-a flexible way to express collective communication. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 574–581. IEEE, 2009.

[61] J. Hofmann, J. Eitzinger, and D. Fey. Execution-cache-memory performance model: Introduction and validation. *arXiv preprint arXiv:1509.03118*, 2015.

[62] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[63] Intel architecture code analyzer homepage. `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer`.

[64] Intel cpu specifications. `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf`.

[65] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *European Conference on Parallel Processing*, pages 196–205. Springer, 2005.

[66] I. T. Jolliffe and J. Cadima. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202, 2016.

[67] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 99–108. IEEE, 2006.

[68] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. *Parallel programming using C+*, pages 175–213, 1996.

[69] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 37–37. ACM, 2001.

[70] T. Kisuki, P. M. Knijnenburg, and M. F. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*, pages 237–246. IEEE, 2000.

[71] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[72] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[73] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[74] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[75] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[76] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 185–194. ACM, 2006.

[77] B. C. Lee and D. M. Brooks. Illustrative design space studies with microarchitectural regression models. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 340–351. IEEE, 2007.

[78] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[79] S. Lee, J. S. Meredith, and J. S. Vetter. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 405–414. ACM, 2015.

[80] S. Lee and J. S. Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 115–120. ACM, 2014.

[81] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 89–100. IEEE, 2009.

[82] R. Lim, B. Norris, and A. Malony. A similarity measure for GPU kernel subgraph matching. *31st International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Oct. 2018.

[83] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou. αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678, 2018.

[84] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 53–64. IEEE, 2009.

[85] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[86] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE, 2009.

[87] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213. Citeseer, 2006.

[88] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[89] J. D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 19:25, 1995.

[90] Mconvolver. `http://www.sdsc.edu/pmac/MetaSim/Convolver/convolver.html`.

[91] K. Meng and B. Norris. Mira: A framework for static performance analysis. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 103–113. IEEE, 2017.

[92] K. Meng and B. Norris. Guiding code optimizations with deep learning-based code matching. In *The 33rd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2020.

[93] Metasim. `http://www.sdsc.edu/pmac/MetaSim/Tracer/tracer.html`.

[94] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[95] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65, 1986.

[96] B. Mohr. Scalable parallel performance measurement and analysis tools-state-of-the-art and future challenges. *Supercomputing frontiers and innovations*, 1(2):108–123, 2014.

[97] A. Monakov, A. Lokhmotov, and A. Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 111–125. Springer, 2010.

[98] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for powerpc microarchitecture exploration. *IEEE Micro*, 19(3):15–25, 1999.

[99] Mpid. `http://www.cepba.upc.es/`.

[100] D. Müller-Wichards. Problem size scaling in the presence of parallel overhead. *Parallel Computing*, 17(12):1361–1376, 1991.

[101] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.

[102] S. H. K. Narayanan, B. Norris, and P. D. Hovland. Generating performance bounds from source code. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 197–206. IEEE, 2010.

[103] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[104] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pages 2014–2023, 2016.

[105] B. Norris, A. Hartono, and W. Gropp. Annotations for productivity and performance portability. In *Petascale Computing: Algorithms and Applications*, Computational Science, pages 443–462. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007. Also available as Preprint ANL/MCS-P1392-0107.

[106] S. Ostermann, R. Prodan, and T. Fahringer. Dynamic cloud provisioning for scientific grid workflows. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 97–104. IEEE, 2010.

[107] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[108] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[109] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, pages 549–562, Austin, TX, Jan. 2011. ACM Press.

[110] S. Prakash and R. L. Bagrodia. Mpi-sim: using parallel simulation to evaluate mpi programs. In *Proceedings of the 30th conference on Winter simulation*, pages 467–474. IEEE Computer Society Press, 1998.

[111] D. Quinlan. ROSE web page, http://rosecompiler.org.

[112] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan. Treating a user-defined parallel library as a domain-specific language. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 324. IEEE Computer Society, 2002.

[113] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood. The structural simulation toolkit: exploring novel architectures. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 157. ACM, 2006.

[114] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[115] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[116] Y. Sakamoto, M. Ishiguro, and G. Kitagawa. Akaike information criterion statistics. *Dordrecht, The Netherlands: D. Reidel*, page 81, 1986.

[117] S. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[118] A. P. Singh and G. J. Gordon. Relational learning via collective matrix factorization. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 650–658, 2008.

[119] R. Smith and P. Gent. Reference manual for the parallel ocean program (pop). *Los Alamos unclassified report LA-UR-02-2484*, 2002.

[120] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 21–21. IEEE, 2002.

[121] O. Source. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org*, 2016.

[122] K. L. Spafford and J. S. Vetter. Aspen: a domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 84. IEEE Computer Society Press, 2012.

[123] SPAPT benchmark codes.
`https://github.com/brnorris03/Orio/tree/master/testsuite/SPAPT`. Last accessed 4/22/2020.

[124] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[125] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*, pages 123–134. IEEE, 2005.

[126] N. R. Tallent and A. Hoisie. Palm: easing the burden of analytical performance modeling. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 221–230. ACM, 2014.

[127] M. Taufer, A. Kerstens, T. Estrada, D. A. Flores, and P. J. Teller. Simba: A discrete event simulator for performance prediction of volunteer computing projects. In *PADS*, volume 7, pages 189–197, 2007.

[128] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[129] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely. Psins: An open source event tracer and execution simulator for mpi applications. In *European Conference on Parallel Processing*, pages 135–148. Springer, 2009.

[130] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[131] B. M. Tudor and Y. M. Teo. A practical approach for performance analysis of shared-memory programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 652–663. IEEE, 2011.

[132] B. M. Tudor, Y. M. Teo, and S. See. Understanding off-chip memory contention of parallel programs in multicore systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 602–611. IEEE, 2011.

[133] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf. Exasat: An exascale co-design tool for performance modeling. *The International Journal of High Performance Computing Applications*, 29(2):209–232, 2015.

[134] J. Vetter and C. Chambreau. mpip: Lightweight, scalable mpi profiling. 2005.

[135] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, 1994.

[136] A. J. Wallcraft, A. B. Kara, H. E. Hurlburt, and P. A. Rochford. The nrl layered global ocean model (nlom) with an embedded mixed layer submodel: Formulation and tuning. *Journal of Atmospheric and Oceanic Technology*, 20(11):1601–1615, 2003.

[137] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM Sigplan notices*, volume 44, pages 75–84. ACM, 2009.

[138] Y. Wen, Z. Wang, and M. F. O'boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[139] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[140] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.

[141] J. Ye, R. Janardan, and Q. Li. Two-dimensional linear discriminant analysis. In *Advances in neural information processing systems*, pages 1569–1576, 2005.

[142] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34. IEEE, 2007.

[143] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 190–199, 2010.

[144] A. Zell, N. Mache, R. Huebner, G. Mamier, M. Vogt, K.-u. Herrmann, M. Schmalzl, T. Sommer, A. Hatzigeorgiou, S. Döring, et al. Snns-stuttgart neural network simulator. 1993.

[145] G. Zheng, G. Kakulapati, and L. V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 78. IEEE, 2004.

[146] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. *International Journal of Parallel Programming*, 33(2-3):183–207, 2005.