Static Program Analysis for Performance Modeling

Kewen Meng, Boyana Norris
Department of Computer and Information Science
University of Oregon
Eugene, Oregon
{*kewen, norris*}*@cs.uoregon.edu*

***Abstract*—The performance model of an application can provide understanding about its runtime behavior on particular hardware. Such information can be analyzed by developers for performance tuning. However, model building and analyzing is frequently ignored during software development until performance problems arise because they require significant expertise and can involve many time-consuming application runs. In this paper, we propose a faster, accurate, flexible and user-friendly tool, Mira, for generating intuitive performance models by applying static program analysis, targeting scientific applications running on supercomputers. Our tool parses both the source 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 non-existent 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 several benchmark validation results to demonstrate the current capabilities of our approach.**

***Keywords*-HPC; performance; modeling; static analysis**

I. Introduction

Understanding application and system performance plays a critical role in supercomputing software and architecture design. Developers require a thorough insight of the application and system to aid their development and improvement. Performance modeling provides software developers with necessary information about bottlenecks and can guide users in identifying potential 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 along with the rise in computing capability, it is also true that many applications cannot use the full available computing potential, which wastes a considerable amount of computing power. 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 in order to gain more information about its bottlenecks and performance potential. Program analysis provides insight on how exactly the instructions are executed in the CPU and data are transferred in the memory, which can be used for further optimization of a program. In my directed research project, we developed an approach for analyzing and modeling programs using primarily static analysis techniques.

Current program performance analysis tools can be categorized into two types: static and dynamic. Dynamic analysis is performed with execution of the target program. Runtime performance information is collected through instrumentation or the hardware counter sampling. PAPI [1] is a widely used platform-independent interface to hardware performance counters. By contrast, static analysis operates on the source code or binary code without actually executing it. PBound [2] is one of the static analysis tools for automatically modeling program performance. It parses the source code and collects operation information from it to automatically generate parameterized expressions. Combined with architectural information, it can be used to estimate the program performance on a particular platform. Because PBound solely relies on

the source code, dynamic behaviors like memory allocation or recursion as well as the compiler optimization would impact the accuracy of the analysis result. However, compared with the high cost of resources and time of dynamic analysis, static analysis is relatively much faster and requires less resources.

While some past research efforts mix static and dynamic analysis to create a performance tool, relatively little effort has been put into pure static performance analysis and increasing the accuracy of static analysis. Our approach starts from object code since the optimization behavior of compiler would cause non-negligible effects on the analysis accuracy. In addition, object code is language-independent and much closer to its runtime situation. Although object code could provide instruction-level information, it still fails to offer some critical factors for understanding the target program. For instance, it is impossible for object code to provide detailed information about code structures and variable names which will be used for modeling the program. Therefore, source code will be introduced in our project as a supplement to collect necessary information. Combining source code and object code, we are able to obtain a more precise description of the program and its possible behavior when running on a CPU, which will improve the accuracy of modeling. The output of our project can be used as a way to rapidly explore detailed information of given programs. In addition, because the analysis is machine-independent, our project provides users valuable insight of how programs may run on particular architecture without purchasing expensive machines. Furthermore, the output of our project can also be applied to create Roofline arithmetic intensity estimates for performance modeling.

This report is organized as follows: Section II briefly describes the ROSE compiler framework, the polyhedral model for loop analysis, and performance measurement and analysis tools background. In Section III, we introduce related work about static performance modeling. In Sections IV and V, we discuss our static analysis approach and the tool implementation. Section VI evaluates the accuracy of the generated models on some benchmark codes. Section VII concludes with a summary and future work discussion.

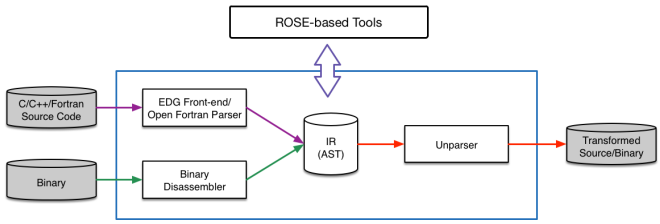


Figure 1: ROSE overview

II. Background

A. ROSE Compiler Framework

ROSE [3] 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. As shown in Figure 1, 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 a number of 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 tools both for experienced compiler researchers and tool developers with minimal background to build custom tools for static analysis, program optimization, and performance analysis.

B. Polyhedral Model

The polyhedral model is an intuitive algebraic representation that treats each loop iteration as 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 is a convex set. The polyhedral model is well suited for optimizing the algorithms with complexity depending on the their structure rather than the number of elements. 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 [4], the polyhedral model is

broadly used for automating optimization and parallelization in compilers (e.g., GLooG [5]) and other tools [6]–[8].

C. Performance Tools

Performance tools are capable of gathering performance metrics either dynamically (instrumentation, sampling) or statically. PAPI [1] is used to access hardware performance counters through both high- and low-level interfaces. The high-level interface supports simple measurement and event-related functionality such as start, stop or read, whereas the low-level interface is designed to deal with more complicated needs. The Tuning and Analysis Utilities (TAU) [9] is another state-of-the-art performance tool that uses PAPI as the low-level interface to gather hardware counter data. TAU is able to monitor and collect performance metrics by instrumentation or event-based sampling. In addition, TAU also has a performance database for data storage and analysis andvisualization components, ParaProf. There are several similar performance tools including HPCToolkit [10], Scalasca [11], MIAMI [12], gprof [13], Byfl [14], which can also be used to analyze application or systems performance.

III. Related Work

There are two related tools that we are aware of designed for static performance modeling, PBound [2] and Kerncraft [15]. Narayanan et al. introduce PBound for automatically estimating upper performance bounds of C/C++ application by static compiler analysis. It collects information and generates parametric expression for particular operations including memory access, floating-point operation, which is combined with user-provided architectural information to compute machine-specific performance estimates. However, it relies on source code analysis, and ignores the effects of compiler behavior (e.g., compiler optimization).

Hammer et al. describe Kerncraft, a static performance modeling tool with concentration on memory hierarchy. It has capabilities of predicting performance and scaling loop behavior based on Roofline [16] or Execution-Cache-Memory (ECM) [17] model. It uses YAML as the file format to describe low-level architecture and Intel Architecture Code Analyzer (IACA) [18] to operate on binaries in order to gather loop-relevant information. However, the reliance on IACA limits the applicability of the tool so that the binary analysis is restricted by Intel architecture and compiler.

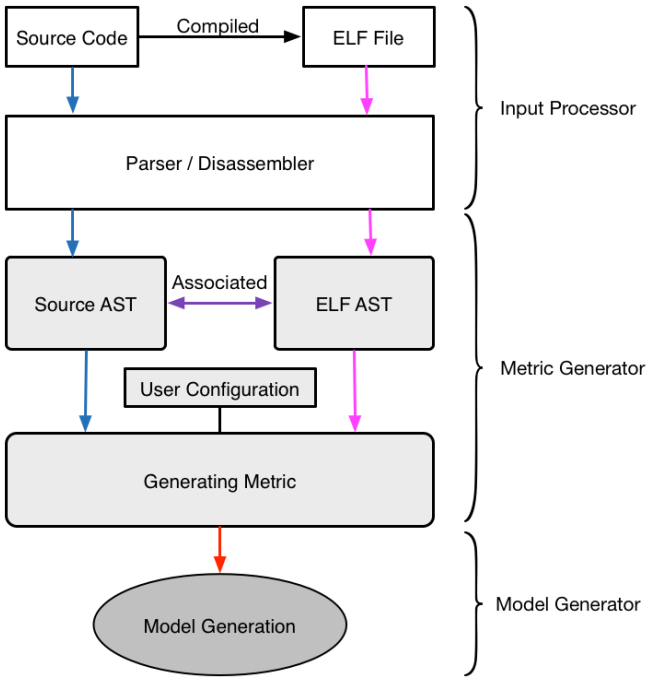


Figure 2: Mira overview

In addition, system simulators can also used for modeling, for example, the Structural Simulation Toolkit (SST) [19]. However, as a system simulator, SST has a different focus—it simulates the whole system instead of signle applications and it analyzes the interaction among architecture, programming model and communications system. Moreover, simulation is computationally expensive and limits the size and complexity of the applications that can be simulated. Compared with PBound, Kerncraft and Mira, SST is relatively heavyweight, complex, and focuses on hardware, which is more suitable for exploring architecture, rather than performance of the application.

IV. Methodology and Implementation

Our Mira static performance modeling tool is built on top of ROSE compiler framework, which provides several useful lower-level APIs as front-ends for parsing the source file and disassembling the ELF file. Mira is implemented in C++ and is able to process C/C++ source code as input. The entire tool can be divided into three parts:

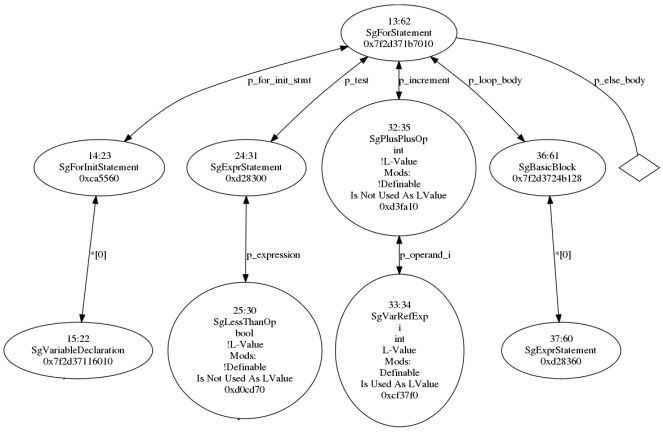


Figure 3: Loop structure in AST

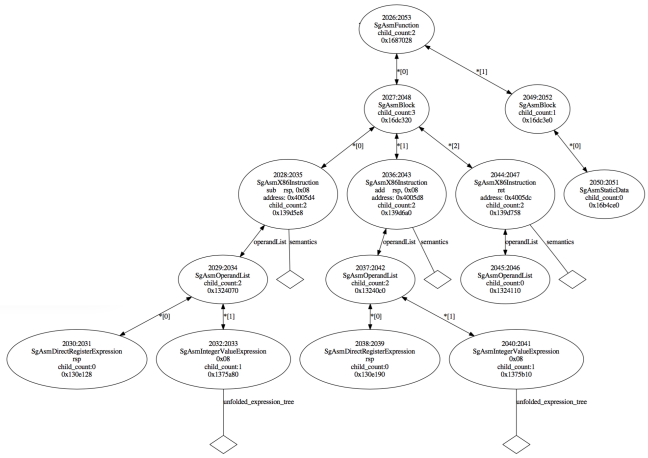


Figure 4: Partial binary AST

- **Input Processor** - Input parsing and disassembling
- **Metric Generator** - AST traversal and metric generation
- **Model Generator** - Model generation in Python

A. Source code and binary representations

The **Input Processor** is the front-end of our modeling tool, and its primary goal is to process input and build an AST (Abstract Syntax Tree). It accepts the source code and ELF file as the input to generating the ASTs which are the representation of the source code and ELF. The AST represents the structure of the program, and enables the modeling tool to locate critical structures such as function bodies, loops, and branches. Furthermore, because the AST also preserves high-level source information, such as variable names, types, the order of statements and the right/left hand side of assignment, it is convenient for the tool to retrieve necessary information and perform analysis. For instance, by knowing the AST, it is possible to query all information about the static control part (SCoP) of the loop to model a loop, including loop initialization, loop condition, and step. In addition, because variable names are preserved, it makes identification of loop indexes much easier and processing of the variables inside the loop more accurate.

B. Bridge between source and binary

1) Difference in AST: The AST is the output of the frond end part of our modeling tool. After processing the inputs, two ASTs are generated separately from source and binary representing the structures of the two inputs. Our tool is designed to use information retrieved from these trees to improve the accuracy of the generated models. Therefore, it is necessary to build connections between the two ASTs so that for a structure in source it is able to instantly locate corresponding nodes in the binary one.

Although both ASTs are representations of the inputs, they have totally different shapes, node organizations and meaning of nodes. A partial binary AST (representing a function) is shown in Figure 4.Each node of the binary AST describes the syntax element of assembly code, such as *SgAsmFunction*, *SgAsmX86Instruction*. As shown in Figure 4, a function in binary AST is comprised of multiple instructions while a functions is made up of statements in source AST, hence it is highly possible that one source AST node corresponds to several nodes in binary AST, which complicates the building of connection between them.

2) Line number: Because the differences between the two AST structures make it difficult to connect source to binary, an alternate 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 source to binary. When we are debugging a program, the debugger knows exactly the source line and column to locate errors. In fact, when using the *-g* option during program compilation, the compiler will insert debug-related information into the object file for future reference. Most compilers

Table I: DWARF Section and its description

Section	Description
.debug_info	The core DWARF information section
.debug_loc	Location lists used in DW_AT_location attributes
.debug_frame	Call frame information
.debug_abbrev	Abbreviations used in the .debug_info section
.debug_aranges	Lookup table for mapping addresses to compilation units
.debug_line	Line number information
.debug_macinfo	Macro information
.debug_pubnames	Lookup table for mapping object and function names to compilation units
.debug_pubtypes	Lookup table for mapping type names to compilation units
.debug_ranges	Address ranges used in DW_AT_ranges attributes
.debug_str	String table used in .debug_info

and debuggers use DWARF (debugging with attributed record format) as the debugging file format to organize the information for source-level debugging. As shown in Table I, DWARF categories data into several sections, such as *.debug_info*, *.debug_frame*, etc. The *.debug_line* section stores the line number information.

Knowing the location of line number information allows us to decode the specific DWARF section to map the line number to the corresponding instruction address. Because line number information in the source AST is already preserved in each node, unlike the binary AST, it can be retrieved directly. After line numbers are obtained from both source and binary, connections are built in each direction. As it is mentioned in the previous section, a source AST node normally links to several binary AST nodes due to the different meaning of nodes. Specifically, a statement contains several instructions, but an instruction only has one connected source location. Once the node in binary AST is associated to the source location, more 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 block such as function body, loop body, and even a single statement.

C. Generating metrics

The **Metric Generator** is an important part of the entire tool, which has significant impact on the accuracy of the generated model. It receives the ASTs as inputs from the Input Processor and to produce parameterized operation counts that are output as Python code. 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 a supplement used for analysis and modeling. For example, if it is too long, one statement is probably located in several lines. In this case, all the line numbers will be collected together and stored as extra information attached to the statement node.

To best model the program, **Model Builder** traverses the source AST twice in different manners, bottom-up and then top-down. The upward traversal propagates detailed information about specific structures up to the head node of the sub-tree. For instance, as shown in Figure 3, *SgForStatement* is the head node for the loop sub-tree; 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 context in modeling.

After upward traversal, top-down traversal is applied to the AST. Because information about sub-tree structure has been collected and attached, the downward traversal primarily focuses on the head node of sub-tree and those of interest, for example the *loop* head node, *if* head node, *function* head node, and *assignment* node, etc. Moreover, it is of significant importance for the top-down traversal to pass down necessary information from parent to child node in order to model complicated structures correctly. For example, in nested loop and branch inside loop the

Table II: Loop coverage in high-performance applications

Application	Number of loops	Number 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%

inner structure requires the information from parent node as the outer context to model itself, otherwise these complicated structures can not be correctly handled. Also, instruction information from ELF AST is connected and associated to correspond structures in top-down traversal.

V. Generating Models

The **Model Generator** is built on the Metric Generator, which consumes the intermediate analysis result of the model builder and generates an easy-to-use model. To achieve the flexibility, the generated model is coded in Python so that the result of the model can be directly applied to various scientific libraries for further analysis and visualization. In some cases, the model is in ready-to-execute condition for which users are able to run it directly without providing any input. However, users are required to feed extra input in order to run the model when the model contains parametric expressions. The parametric expression exists in the model because our static analysis is not able to handle some cases. For example, when user input is expected in the source code or the value of a variable comes from the returning of a call, the variables are preserved in the model as parameters that will be specified by the users before running the model.

A. Loop modeling

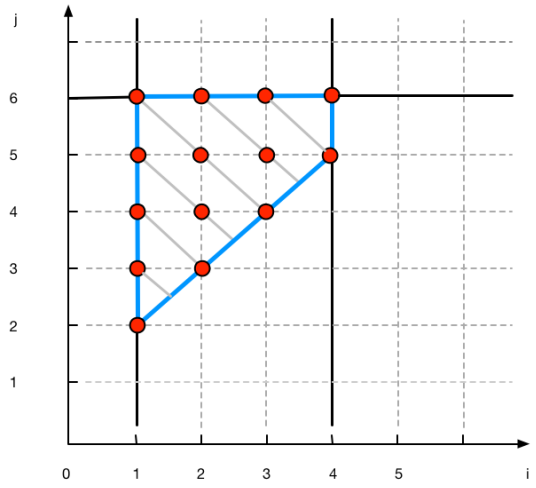
Loops are common in HPC codes and are typically at the heart of the most time-consuming computations. A loop executes a block of code repeatedly until certain conditions are satisfied. Bastoul et al. [20] surveyed multiple high-performance applications and summarized the results in Table II. The first 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 are 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 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.

B. Using the polyhedral model

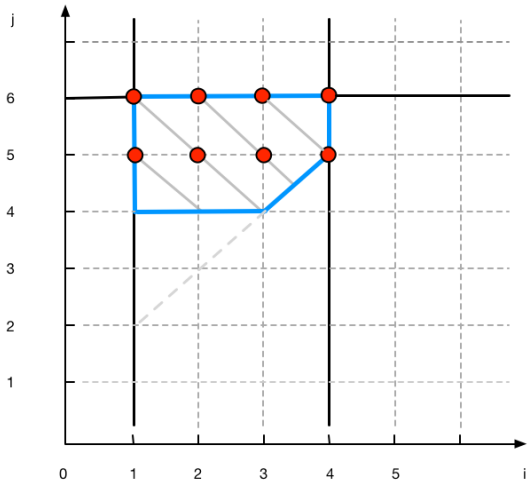
Listing 1: Basic loop

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

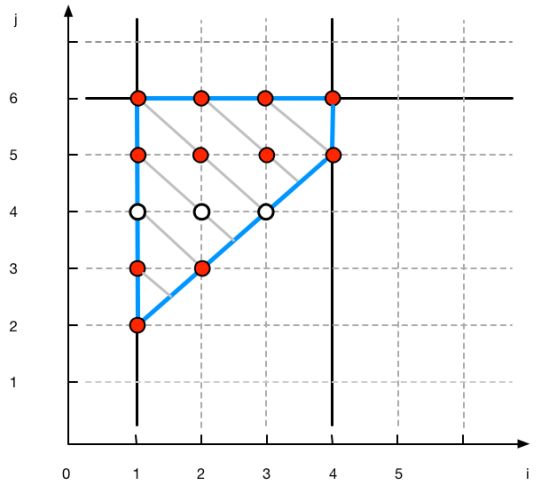
Whether loops can be precisely described and modeled has a direct impact on the accuracy of the generated model because the information about loops will be provided as context for further in-loop analysis. The term “loop modeling” refers to analysis of the static control parts (SCoP) of a loop to obtain the information about the loop iteration domain, which includes understanding of the initialization, termination condition and step. Unlike dynamic analysis tools which may collect runtime information during execution, our approach runs statically so the loop modeling primarily relies on SCoP parsing and analyzing. Usually to model a loop, it is necessary to take several factors into consideration, such as depth, variable dependency, bounds, etc. Listing 1 shows a basic loop structure, the SCoP is complete and simple without any unknown variable.



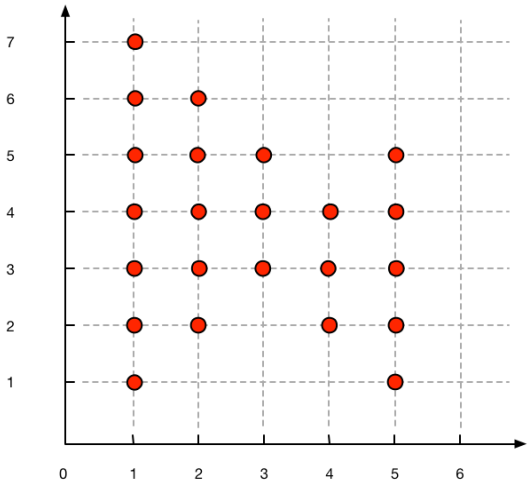
(a) Polyhedral representation for double-nested loop



(b) Polyhedral representation with *if* constraint



(c) *if* constraint causing holes in the polyhedral area



(d) Exceptions in polyhedral modeling

Figure 5: Modeling double-nested loop

For this case, it is possible to retrieve the initial value, upper bound and steps 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 the from binary AST for the statements in Listing 1, the actual count of these instructions is expected to be multiplied by the iteration count to describe the real situation during runtime.

Listing 2: Double-nested loop

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

However, loops in real application are more complicated, which requires our tool to handle as many scenarios as possible. Therefore, the challenge for modeling the loop is to create a general method for various cases. To address this problem, we use the polyhedral model in Mira to accurately model the loop. The polyhedral model is capable of handling an N-

dimensional nested loop and represents the iteration domain in an N-dimensional polyhedral space. For some cases, the index of inner loop has a dependency with the outer loop index. As shown in Listing 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 time-consuming. 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 $1 \leq i \leq 4$ and $i+1 \leq j \leq 6$, which can be written as two equations separately:

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

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

In Figure 5(a), the two-dimensional polyhedral area presenting 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 integer in the polyhedral space, we are able to parse the loop iteration domain and obtain the iteration times. Besides, for loops with more complicated SCoP, such as the ones contain variables instead of concrete numerical values, the polyhedral model is also able to handle. 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 as 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 re-generate the model for different values of the parameters. Instead, they just have to adjust the inputs for the model and run the Python to produce a concrete value.

Listing 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 that the polyhedral model cannot handle. For the code snippet in Listing 3, the SCoP of the loop forms a non-convex set (Figure 5(d)) to which polyhedral model can not apply. Another problem in this code is that the loop initial value and loop bound depend on the returning value of function calls. For static analysis to track and obtain the returning value of function calls, more complex interprocedural analysis is required. For such scenarios, we need more sophisticated approach that will be a part of our future work.

C. Branches

Listing 4: Loop with *if* constraint

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

In addition to loops, branch statements are common structures. In scientific applications, branch statements are frequently used to verify the intermediate output during the computing. Branch statements can be handled by the information retrieved from the AST. However, it complicates the analysis when the branch statements reside in a loop. In Listing 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 5(b), 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 5(b) with Figure 5(a), 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 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 that the polyhedral model is not applicable. For the code in Listing 5, the *if* condition excludes several integers in the polyhedral space causing "holes" in the iteration space as shown in Figure (c). The excluded integers break the integrity of the space so that it no longer satisfies the definition of the convex set, thus the polyhedral model is not available for this particular circumstance. In this case, the true branch of the *if* statement raises the problem but the false branch still satisfies the polyhedral model. Thus we can use:

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

Since the counter of the outer loop and false branch both can be expressed by the polyhedral model, using either concrete value or parametric expression, so the count of the true branch is obtained. 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 model or even static analysis, such as loop-index irrelevant variables or function call located inside the branch condition. For such circumstances, we provide users an option to annotate branches inside loops. The procedure of annotation requires users to give an estimated percentage that the branch may take in the particular loop iteration or indication of skipping the branch. The annotation is written in the source code as a comment, and processed during the generation of the AST.

Listing 6: User annotation for *if* statement

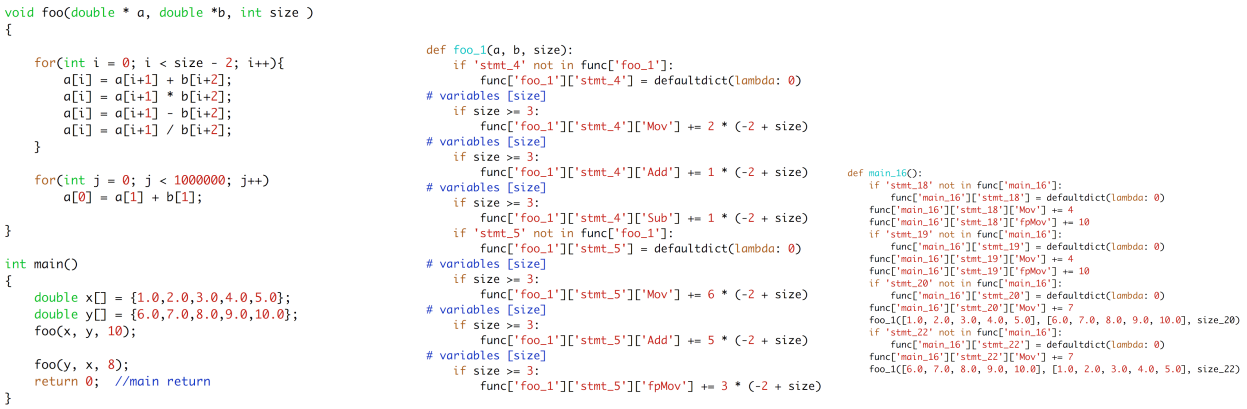
```
for(i = 1; i <= 4; i++)
 for(j = i + 1; j <= 6; j++)
 {
  if(foo(i) > 10)
  /* @Annotation {p:skip} */
  {
   statements;
  }
 }
```

In order to improve the applicability, an option is provided to the users for annotation of the branch that is not able to handle by Mira. As the example shown in Listing 6, the *if* condition contains a function call which cannot be modeled by Mira at present because it is difficult for a static tool to obtain the returning value of a function call. To address such problem, users could specify an annotation in the comment to guide Mira's modeling. In the given example, the whole branch will be skipped when using polyhedral model. In addition to skip of the branch, a numeric value in the annotation represents the percentage of iteration times of this branch in the whole loop. For instance, *@Annotation {p:0.8}* means the body of the branch is executed 80% of the total iterations of the loop.

D. Functions

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 makes a minor modification in the function name in order to avoid potential conflict due to function overwrite or other special cases. In the body of the generated Python function, statements are replaced with corresponding performance metrics data retrieved from binary. These data are stored in Python dictionaries and organized in the same order as the statements. Each function is executable and returns the aggregate result within its scope. The advantage of this design is to provide the user the freedom to separate and obtain the overview of the particular functions with only minor changes to the model.

Another design goal is to appropriately handle function calls. For modeling function calls, because all of the user-defined functions are generated in Python



```
void foo(double * a, double *b, int size )
{

    for(int i = 0; i < size - 2; i++){
        a[i] = a[i+1] + b[i+2];
        a[i] = a[i+1] * b[i+2];
        a[i] = a[i+1] - b[i+2];
        a[i] = a[i+1] / b[i+2];
    }

    for(int j = 0; j < 1000000; j++)
        a[0] = a[1] + b[1];

}

int main()
{
    double x[] = {1.0,2.0,3.0,4.0,5.0};
    double y[] = {6.0,7.0,8.0,9.0,10.0};
    foo(x, y, 10);

    foo(y, x, 8);
    return 0;  //main return
}
```

(a) Source code

```
def foo_1(a, b, size):
    if 'stmt_4' not in func['foo_1']:
        func['foo_1']['stmt_4'] = defaultdict(lambda: 0)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_4']['Mov'] += 2 * (-2 + size)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_4']['Add'] += 1 * (-2 + size)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_4']['Sub'] += 1 * (-2 + size)
    if 'stmt_5' not in func['foo_1']:
        func['foo_1']['stmt_5'] = defaultdict(lambda: 0)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_5']['Mov'] += 6 * (-2 + size)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_5']['Add'] += 5 * (-2 + size)
# variables [size]
    if size >= 3:
        func['foo_1']['stmt_5']['fpMov'] += 3 * (-2 + size)
```

(b) Generated *foo* function

```
def main_16():
    if 'stmt_18' not in func['main_16']:
        func['main_16']['stmt_18'] = defaultdict(lambda: 0)
    func['main_16']['stmt_18']['Mov'] += 4
    func['main_16']['stmt_18']['fpMov'] += 10
    if 'stmt_19' not in func['main_16']:
        func['main_16']['stmt_19'] = defaultdict(lambda: 0)
    func['main_16']['stmt_19']['Mov'] += 4
    func['main_16']['stmt_19']['fpMov'] += 10
    if 'stmt_20' not in func['main_16']:
        func['main_16']['stmt_20'] = defaultdict(lambda: 0)
    func['main_16']['stmt_20']['Mov'] += 7
    foo_1([1.0, 2.0, 3.0, 4.0, 5.0], [6.0, 7.0, 8.0, 9.0, 10.0], size_20)
    if 'stmt_22' not in func['main_16']:
        func['main_16']['stmt_22'] = defaultdict(lambda: 0)
    func['main_16']['stmt_22']['Mov'] += 7
    foo_1([6.0, 7.0, 8.0, 9.0, 10.0], [1.0, 2.0, 3.0, 4.0, 5.0], size_22)
```

(c) Generated *main* function

Figure 6: Generated Model

but with modified names, the first step is to search the function-name mapping record maintained in the model and obtain the modified name. For instance Python function with name *foo_223* represents that the original name is *foo* and location in the source code is line 223. Since the callee function from the source named as *foo*, it is necessary for the callee function to recover its original name. Then in the function signature, the actual parameters are retrieved from the source AST and combined with function name to generate a function call statement in the Python model.

E. Architecture configuration file

Because Mira provides a parameterized performance model of the performance of the target application running on different architectures and no architectural information is collected during modeling, it is necessary for the users to provide information about the target architectures, such as number of CPU cores, size of cache line, and vector length, etc (in future, we will use microbenchmarks to obtain these values automatically). For instance, in data-parallel (SIMD) processors, it is of importance to know how much data is processed simultaneously. These hardware-related metrics are used to describe the target architectures, and Mira reads them from a user configurable file to apply to the model evaluation. By giving the detail information of the hardware, it improves the accuracy and extendability of the generated model.

F. Generated model

In this section, we describe the model generated (output) by Mira with an example. In Figure 6, it shows the source code (input) and generated Python model separately. Source code (Figure 6(a)) contains two functions: function *foo* with three parameters including two array variables and one integer used as loop upper bound. The *main* function is the entry point of the whole program. In addition, the function *foo* is called two times in *main* with different values for parameter *size*. Figure 6(b) shows part of the generated Python function *foo* in which the new function name is comprised of original function name and its line number in order to avoid possible naming conflicts. The body of the generated function *foo_1* contains Python statements for keeping track of performance metrics. Similarly, the *main* function is also modeled shown as Figure 6(c) which includes two function calls. It is noted that the only difference between the two function calls is the third parameter. Since Mira recognizes that *size* is a loop-relevant variable, therefore it should be preserved as a parameter in the generated model whose value can be specified by the user. Two parameters in the function calls named differently because function call is separate to each other in the generated model.

G. Evaluating the model

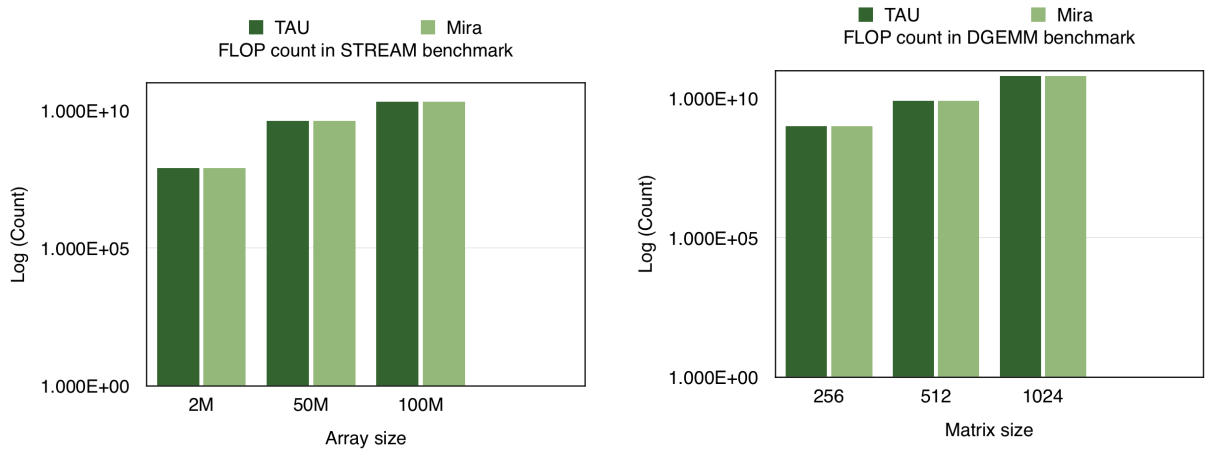
In this section, we evaluate the correctness of the model derived by our tool with TAU in instrumentation mode. Two benchmarks are separately executed statically and dynamically on two different machines. Since

Table III: FLOP count in STREAM benchmark

Tool / Array size	2M	50M	100M
TAU	8.239E7	4.108E9	2.055E10
Mira	8.20E7	4.100E9	2.050E10

Table IV: FLOP count in DGEMM benchmark

Tool / Matrix size	256	512	1024
TAU	1.013E9	8.077E9	6.452E10
Mira	1.0125E9	8.0769E9	6.4519E10



(a) FLOP count in STREAM benchmark with different array size(b) FLOP count in DGEMM benchmark with different matrix size

Figure 7: Validation results

the our performance tool currently concentrates on floating-point operations, the validation is performed on comparison of the floating-point operation counting between our tool and TAU measurements.

1) Experiment environment: Mira is able to generate performance model of the target machines with different architectures. Therefore the validation is conducted on two different machines. The details of the two machines are as follows:

- **Arya** - It is equipped with two Intel Xeon E5-2699v3 2.30GHz 18-core Haswell micro-architecture CPUs and 256GB of memory.
- **Frankenstein** - It has two Intel Xeon E5620 2.40GHz 4-core Nehalem micro-architecture CPUs and 22GB of memory.

2) Benchmarks: Two benchmarks are chosen for validation, STREAM and DGEMM. 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.

VI. Results

In this section, we present the validation results and discuss the primary features in on our tool which have significant impact on users for evaluating the tradeoff between static and dynamic methods for performance analysis and modeling.

Table III and IV shows the results of FLOP counting in two benchmarks separately. The metrics are gathered by the calculation of the model generated by our tool, and instrumentation by TAU. In Figure 7(a), the X axis is the size of the array which represents the

problem size, and we choose 20 million, 50 million and 100 million respectively as the inputs. To make the graph clear enough, the FLOP counts is scaled logarithmically and shown on the Y axis. Similarly, in Figure 7(b), the X axis is for input size and the Y for FLOP counts. From the tables and graphs, we can see the result from our model is close to the one instrumented by TAU. Moreover, the difference of results increases along with the growth of problem size. It is possibly due to the function calls inside of loops. TAU conducts a whole program instrumentation, which records every floating-point operation in every routine of the program. However, our static modeling tool is not able to handle non-user-defined function calls, such as a function defined in mathematical library. For such scenarios, Mira can only track the function call statements that may just contain several stack manipulation instructions while the function itself involves the floating-point operations.

Besides correctness, we compare the execution time of the two methods. To analyze a program, the source code must be recompiled after adjustment of the inputs. Because the analysis works on every single instruction, it spends a large amount of time on uninteresting instructions, thus it can be time-consuming. However, our model only needs to be generated once, and then can be used to obtain the corresponding results based on different inputs, which saves many efforts and avoids duplicated work. Most importantly, the performance analysis by a parametric model achieves higher coverage than repeatedly conducting of the experiments.

Furthermore, availability is a another factor needed to take into account. Due to the changes on the later models of Intel CPUs, PAPI-based performance tools are not able to gather some particular performance metrics (e.g. FLOP counts) by hardware counters. Hence, static performance analysis may be a solution to such cases.

VII. Conclusion

In this report, we present our work about performance modeling by static analysis. We aim at designing a faster, accurate and flexible method for performance modeling as a supplement to existing tools in order to address problems that cannot solved by current tools. Our method focuses on floating-point operations and achieves good accuracy for benchmarks. These preliminary results suggest that this can be an effective method for performance analysis.

However, much work remains to be done. In our future work, the first problem we eager to tackle is to appropriately handling of more function calls, especially those from system or third-party libraries. We also consider combining dynamic analysis and introducing more performance metrics into the model to accommodate more complicated scenarios. To enhance the scalability is also a significant mission for us in order to enable Mira to model the programs running in the distributed environment.

References

[1] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, 1999, pp. 7–10.

[2] 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*. IEEE, 2010, pp. 197–206.

[3] D. Quinlan, "Rose homepage," http://rosecompiler.org.

[4] 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)*. Austin, TX: ACM Press, Jan. 2011, pp. 549–562.

[5] 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*, Juan-les-Pins, France, September 2004, pp. 7–16.

[6] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures," 2004.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 101–113.

[8] T. Grosser, A. Groesslinger, and C. Lengauer, "Pollyperforming polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

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

[10] 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*, vol. 22, no. 6, pp. 685–701, 2010.

[11] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[12] G. Marin, J. Dongarra, and D. Terpstra, "Miami: A framework for application performance diagnosis," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 158–168.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[14] S. Pakin and P. McCormick, "Hardware-independent application characterization," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 111–112.

[15] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein, "Automatic loop kernel analysis and performance modeling with kerncraft," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015, p. 4.

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

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

[18] Intel, "Intel architecture code analyzer homepage," https://software.intel.com/en-us/articles/intel-architecture-code-analyzer.

[19] SNL, "Sst homepage," http://sst-simulator.org.

[20] 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*. Springer, 2003, pp. 209–225.