LLNL-CONF-868729

Reductive Analysis with Compiler-Guided Large Language Models for Input-Centric Code Optimizations

X. Wang, X. Hui, C. Liao, X. Shen

August 26, 2024

ACM SIGPLAN Conference on Programming Language Design and Implementation
Seoul, South Korea
June 16, 2025 through June 20, 2025

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Reductive Analysis with Compiler-Guided Large Language Models for Input-Centric Code Optimizations

XIANGWEI WANG, North Carolina State University, USA
XINNING HUI, North Carolina State University, USA
CHUNHUA LIAO, Lawrence Livermore National Laboratory, USA
XIPENG SHEN, North Carolina State University, USA

Input-centric program optimization aims to optimize code by considering the relations between program inputs and program behaviors. Despite its promise, a long-standing barrier for its adoption is the difficulty of automatically identifying critical features of complex inputs. This paper introduces a novel technique, *reductive analysis through compiler-guided Large Language Models (LLMs)*, to solve the problem through a synergy between compilers and LLMs. It uses a reductive approach to overcome the scalability and other limitations of LLMs in program code analysis. The solution, for the first time, automates the identification of critical input features without heavy instrumentation or profiling, cutting the time needed for input identification by 44× (or 450× for local LLMs), reduced from 9.6 hours to 13 minutes (with remote LLMs) or 77 seconds (with local LLMs) on average, making input characterization possible to be integrated into the workflow of program compilations. Optimizations on those identified input features show similar or even better results than those identified by previous profiling-based methods, leading to optimizations that yield 92.6% accuracy in selecting the appropriate adaptive OpenMP parallelization decisions, and 20-30% performance improvement of serverless computing while reducing resource usage by 50-60%.

CCS Concepts: • **Software and its engineering** → *Software notations and tools.*

Additional Key Words and Phrases: Large Language Models, Program Optimization, Input-Centric Optimization, Seminal Behavior Identification, Predictive Modeling

ACM Reference Format:
Xiangwei Wang, Xinning Hui, Chunhua Liao, and Xipeng Shen. 2025. Reductive Analysis with Compiler-Guided Large Language Models for Input-Centric Code Optimizations. *Proc. ACM Program. Lang.* 9, PLDI, Article 179 (June 2025), 25 pages. https://doi.org/10.1145/3729282

1 Introduction

For a given runtime environment (hardware, OS, etc.), the runtime behavior and performance of a program are determined by its code and its *input*—which refers to values that the program receives from outside, including its command line arguments, the contents of files it accesses, environment variables it uses, network packets it receives, and so on. Although the compiler community has extensively studied how to optimize the code of a program, how to empower the programming systems to systematically handle program inputs and their influence on a given program remains an open question.

Previous research has devoted some efforts into the problem. The one closest to giving a systematic solution is the *input-centric program optimization* work by Tian and others [33]. The idea

Authors' Contact Information: Xiangwei Wang, North Carolina State University, Raleigh, USA, xwang258@ncsu.edu; Xinning Hui, North Carolina State University, Raleigh, USA, xhui@ncsu.edu; Chunhua Liao, Lawrence Livermore National Laboratory, Livermore, USA, liao6@llnl.gov; Xipeng Shen, North Carolina State University, Raleigh, USA, xshen5@ncsu.edu.

ACM 2475-1421/2025/6-ART179
https://doi.org/10.1145/3729282

is to build up predictive models and integrate them into the program such that at runtime, those models can, based on the critical features of the current inputs, predict how the program is going to behave in this current execution and conduct appropriate runtime optimizations. The previous experiments have demonstrated that predictions from the models can help improve the selection of important functions to do deep Just-In-Time compilations [34], the selection of the more suitable Garbage Collection [22], GPU code optimizations [41], and so on.

Despite the promise of input-centric program optimization, its adoption in practice faces a major barrier: identifying the critical features of the inputs to a program, which is called *input characterization problem*. Program inputs may have arbitrary structures and semantics, ranging from a graph to an audio to a database or even a program. It may have a large number of attributes, from as simple as the values of some special numbers in a file to as complex as the density of a graph, the frequency range of an audio signal, the distribution of a bag of data, and the numbers of various constructs in a program.

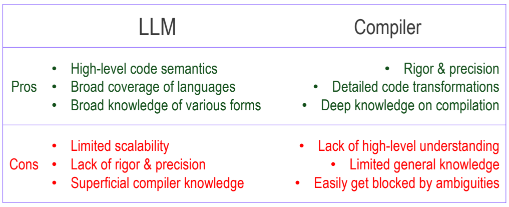
Existing approaches are either resorting to manual efforts [31] or statistical methods [14, 33]. The former not only adds lots of burden on programmers but also prevents the construction of an automatic end-to-end workflow for input-centric optimizations. The latter, known as *profiling-based method*, requires detailed profiling runs of the program of interest on hundreds of different inputs. Through the profiling runs, it collects the counts of the calling frequencies of each function, the tripcounts of every loop, and many other values in the program. It then runs statistical correlation analysis on those counts and values to identify the input features that critically correlate with those behaviors. The *profiling-based method* is tedious and time-consuming. The detailed profiling requires detailed code instrumentation, which typically slows down the execution of the program by several times. The total profiling time for a program may take as much as 44 hours (on program "x264", detailed in Section 6), making the workflow for input-centric optimizations extremely cumbersome, a main reason for the difficulties for the practical adoption of input-centric optimizations.

To seamlessly incorporate input characterization into the workflow of a compiler, which is crucial for making input-centric code optimizations adoptable in practice, the solution must be automatic and lightweight without requiring manual efforts or lengthening the compilation process by hours. Neither of the existing approaches meet the requirement.

In this work, we propose to address the long-standing barrier to automatic input characterization of programs through compiler-guided Large Language Models (LLMs). Note that to figure out what features of inputs are crucial to a program's behaviors (e.g., running time), the place to look at is the program rather than the inputs: It is the program that determines which parts of the inputs are read and how they are used. Therefore, the input characterization problem can be treated as a program analysis problem.

LLM has shown a remarkable capability to digest source code and answer questions about it. It has some advantages over traditional compilers in program analysis, but is also subject to some limitations. Figure 1 summarizes their pros and cons. Most notable is that LLM can capture high-level code semantics but faces scalability limitations in dealing with large codebases and giving reliable code analysis results. The basic **hypothesis** of this work is that their combination, in form of compiler-guided LLM, provides the solution to automatic agile input characterization.

The key technique we propose is reductive analysis with compiler-guided LLM (RACL). RACL is designed to address the scalability limitation of LLMs. Here, the scalability limitation has two levels of meanings. The first is about code size. Current LLMs have a limit on the maximal number of tokens for a request. Although the limit is being continuously raised, there is still a limit, and even if the code size fits in the limit, the quality and speed of LLM analysis still suffer if a large codebase is provided to LLM all together for a request (especially when it is compound with the behavior complexity described next). In the general LLM area, techniques (e.g., retrieval augmented



	LLM	Compiler
Pros	• High-level code semantics • Broad coverage of languages • Broad knowledge of various forms	• Rigor & precision • Detailed code transformations • Deep knowledge on compilation
Cons	• Limited scalability • Lack of rigor & precision • Superficial compiler knowledge	• Lack of high-level understanding • Limited general knowledge • Easily get blocked by ambiguities

Fig. 1. The pros and cons of LLM and compilers in program analysis.

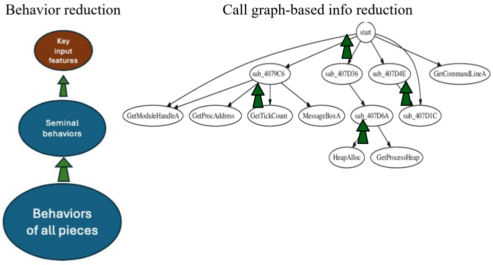


Fig. 2. Reductive strategy employed by RACL

generation [11] and map-reduce [16]) have been developed to go around the limit by retrieving the tokens most relevant to a request. But those techniques, designed for natural languages, frequently leave out important program structure information (e.g., calling relations between functions) when being applied to program analysis, causing poor results (detailed Section 6).

The second meaning of the scalability limitation of LLMs is about program behaviors. The purpose of input characterization is to figure out the features that determine program behaviors. One type of program behavior (e.g., running time) is often composed of the behaviors of almost all pieces of the program, from the executions of every instruction, to the behaviors at every branch, loop, function, and so on. The behavior complexity grows as program code size increases.

Because of the two-level complexities, it is inadequate to directly ask LLM to do input characterization for a non-trivial program. Moreover, even if LLMs can provide solutions to the full program, it is often necessary to also know the key features determining the behaviors of each part of the program for fine-grained analysis and optimizations.

RACL solves the challenges by taking a reductive strategy as shown in Figure 2. The strategy consists of complexity reductions in two dimensions. (i) The first dimension is in program behaviors. The RACL-enabled analysis reduces the focus of analysis from the many behaviors of program components to a much smaller set of behaviors named *seminal behaviors*, and then maps seminal behaviors to program inputs. Seminal behavior is a concept introduced in prior work [14], referring to behaviors in a program that a small set of behaviors that strongly correlate with most other behaviors in the program, and meanwhile, expose their values early in typical executions. The reduction in this dimension reduces the difficulties of directly dealing with the many behaviors when solving the input characterization problem. (ii) The second dimension is in code complexity. RACL uses *modified call graph* to guide LLM to conduct seminal behavior analysis on one function at a time, and importantly, it ensures that the LLM-based analysis of a function take into consideration of the key information of the functions it calls directly or indirectly. It achieves that by creating a representation of the key info of a function and a program, breaking recursion-caused cycles in call graphs without losing key info, and building a mechanism to propagate the key info from callees to callers with proper info translation to keep the info always understandable and prevent its size from exploding. By reducing the focus of LLM to one function each time, RACL makes the analysis free of the token limit of LLM. It meanwhile makes the analysis output the seminal behaviors for each function as a side product.

To demonstrate the potential of RACL for enabling automatic agile input-centric program optimizations, we create LLM-enabled reductive input characterising tool (MERIC), the first end-to-end agile framework that automatically characterizes important features of the inputs to certain behaviors[1] of a program. To ease the use of the characterization results, MERIC in addition includes a tool that employs LLM to generate code that can extract the key feature values from an input to a program.

To test the efficacy of MERIC, we apply it to ten programs in various domains including real-world applications as large as 96K lines of source code. Compared to previous profiling-based methods [14, 33], MERIC cuts the time needed for input identification by 44× (or 450× for local LLMs deployment), reduced the time from 9.6 hours to 13 minutes (or 77 sec for local LLMs) on average, making input characterization possible to be integrated into the workflow of program compilations. We test the usefulness of the characterized input features by MERIC in three uses, runtime optimizations of OpenMP parallelism, shortest job scheduling, and serverless computing. For each use, we build a machine learning model that uses the characterized input features as model input and outputs the prediction of the best parallel configurations or running time of the program of interest. The results show that the input features identified by MERIC lead to similar or even better results than those identified by previous profiling-based methods. The predictive models can attain an average accuracy of 92.6% for OpenMP parallelism. When applied to serverless computing, their predictions make the controllers in an open-source state-of-the-art serverless platform save 50-60% resource usage while delivering 20-30% better performance.

Overall, this paper makes the following contributions:

- It presents the first compiler-guided LLMs for automatic input characterization of programs without extensive profiling or manual analysis.
- It proposes a novel RACL scheme for addressing the scalability issues of LLMs for input characterization, which is potentially applicable to other code analysis problems as well.
- It contributes the first end-to-end tool MERIC and empirically confirms the efficacy of the proposed techniques.

2 A Running Example

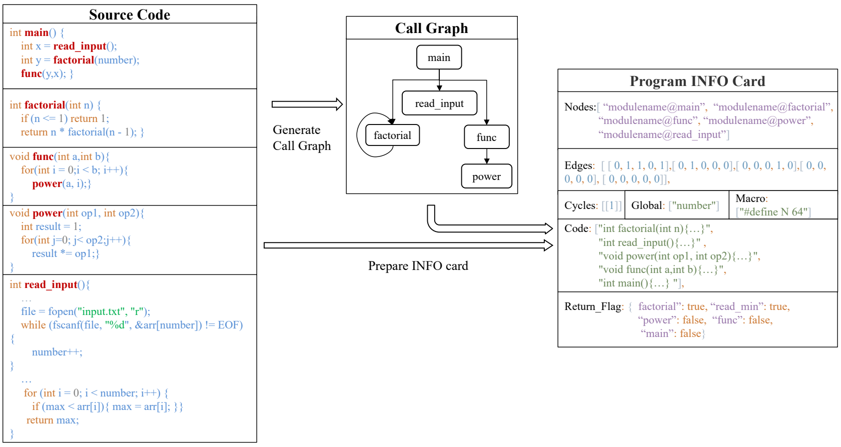
To help the following explanation, we provide a running example as shown in Figure 3. The example is made simple for illustrative purposes. Its input is a list of numbers. Its *main* function calls three functions: *read_input*, *factorial* and *func*. Function *read_input* reads an outside file. This function records how many elements in the file, assigns the number to a global variable, and returns the max value in the file. Function *factorial* is a self-recursive function. It calculates the factorial of the value of a global variable "number". The value of "number" is the number of elements in the input file. Function *func* has a loop, whose upperbound is the second parameter of *func*. This parameter gets its value from the largest number in the input file in function *main*. Inside the loop, it calculates "a" to the power of "i" by calling the function *power*. It can be seen that the input features that critically determine the running time of this program are the number of integers and the max value in the input file. In the following sections, we will use this example to explicitly demonstrate how our MERIC method transforms the source code and identify the program's input features.

3 MERIC Overview and Program INFO Card

The overall workflow of MERIC offers a view of the high-level process of using RACL for input characterization.

As shown in Figure 4, MERIC consists of three main components: *Preparator*, *Code Analyzer* and *Extraction Module Creator*. Through a compiler-based tool (doxygen [8]), the *preparator* in MERIC

[1] One of the most typical program behavior is the program's running time for its relevance to many program optimizations. It is what the discussions in the paper will assume. But other behaviors—such as code size, memory footprint—can also be taken as the target.



Source Code

```
int main() {
   int x = read_input();
   int y = factorial(number);
   func(y,x); }

int factorial(int n) {
   if (n <= 1) return 1;
   return n * factorial(n - 1); }

void func(int a,int b){
   for(int i = 0;i < b; i++){
      power(a, i);}
}

void power(int op1, int op2){
   int result = 1;
   for(int j=0; j< op2;j++){
      result *= op1;}
}

int read_input(){
   ...
   file = fopen("input.txt", "r");
   while (fscanf(file, "%d", &arr[number]) != EOF)
{
      number++;
}
   ...
    for (int i = 0; i < number; i++) {
      if (max < arr[i]){ max = arr[i]; }}
     return max;
}
```

Program INFO Card

```
Nodes:[ “modulename@main”,  “modulename@factorial”,
       “modulename@func”, “modulename@power”,
       “modulename@read_input”]
Edges:  [ [ 0, 1, 1, 0, 1],[ 0, 1, 0, 0, 0],[ 0, 0, 0, 1, 0],[ 0, 0,
0, 0, 0], [ 0, 0, 0, 0, 0]],
Cycles: [[1]]   Global: ["number"]   Macro:
                                     ["#define N 64"]
Code: ["int factorial(int n){…}",
       "int read_input(){…}" ,
       "void power(int op1, int op2){…}",
       "void func(int a,int b){…}",
       "int main(){…} "],
Return_Flag: {  factorial”: true, “read_min”: true,
                “power”: false,  “func”: false,
                 “main”: false}
```

Fig. 3. The source code of a running example. The preparator generates the call graph and gathers the essential information from source code to create program INFO card.

gathers the essential information from the source code and organizes the info into a JSON file, named *program INFO card*. The *code analyzer* is the main component, taking the program INFO card as input and outputting the characterized input features. Inside, the *code analyzer* traverses the reversed (modified) call graph by following the RACL scheme. It gets the seminal behaviors of the functions one by one by leveraging LLMs on the carefully designed prompts, passes the key insights from callees to callers, and eventually maps the important seminal behaviors to program inputs to identify the important input features. The last component, *extraction module creator*, generates a module that can extract the values of the key input features from inputs to the program.

As an essential data structure, the program INFO card contains two parts: *call graph* and *auxiliary info*. In the example in Figure 3, the call graph is shown by the "Nodes" and "Edges" (adjacency matrix) fields.

The other fields are auxiliary fields, containing the supplementary information of the program that are useful for automatically instantiating some prompts templates into prompts to LLMs. The fields are as follows:

- “code”: It is a list to store each function’s source code. It allows the code analyzer to efficiently retrieve the relevant code snippets for each function.
- “cycle”: It records the recursions in the call graph. We employ a common Depth-First Search (DFS)-based algorithm to identify recursions on call graphs. In Figure 3, this field is [[1]], incidating that there is only one recursion in the program and it is a self recursion on node[1] (function `factorial`).
- “global”: It is a list containing all global variables in the program.
- “macro”: It presents useful macros outside the functions.
- “return_flag”: It is a dictionary with function names as keys and boolean values, indicating whether a function has a return value.

We next focus on the key component in MERIC, the RACL code analyzer.

4 RACL Code Analyzer

The RACL code analyzer in MERIC takes program INFO cards as input and outputs the characterized input features obtained through program structure-guided LLMs. It uses a reductive scheme to

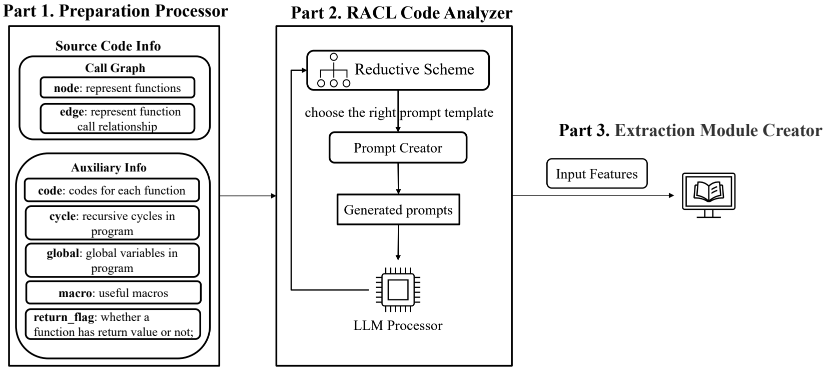

Part 1. Preparation Processor
Source Code Info
Call Graph
node: represent functions
edge: represent function call relationship
Auxiliary Info
code: codes for each function
cycle: recursive cycles in program
global: global variables in program
macro: useful macros
return_flag: whether a function has return value or not;
Part 2. RACL Code Analyzer
Reductive Scheme
choose the right prompt template
Prompt Creator
Generated prompts
LLM Processor
Part 3. Extraction Module Creator
Input Features

Fig. 4. The overall workflow of MERIC.

propagate key info throughout the call graph, so that it can circumvent the scalability limitations of LLMs for code analysis. Its design faces the following difficulties: complexities of recursions in a call graph, the influence of parameter passing and return values, the requirement of LLMs to use high-quality prompts to perform, and the token limit of LLMs. In this section, we present the overall reductive algorithm of the RACL analyzer first, and then explain our treatment of those difficulties.

4.1 Overall Algorithm

Algorithm 1 RACL-based Code Analysis

```
Input: Program INFO card C which contains Function Call Graph G including Node set N and Edge matrix E, Recursion Cycle cycles, and
    other info
Output: Analysis Result Set AnalyRes
 1: funInfo = initFunInfo(C);
 2: G, TerminationCond = EliminateCycle(G);
 3: for T in TerminationCond do
 4:     node = T['function']
 5:     Appendinfo(funInfo[node],T);
 6: end for
 7: Q = FindLeafNodes(G)
 8: cnode = Q.headnode
 9: AnalyRes = ∅
10: while N-Q do
11:     AnalyRes[cnode] = GetAnalyRes(cnode, funInfo[cnode]);
12:     for node in N-Q do
13:         if E(node, cnode) == 1 then
14:             Appendinfo(funInfo[node],AnalyRes[cnode]);
15:             node.count++;
16:             if node.count == node.outdegree then
17:                 Q.append(node)
18:             end if
19:         end if
20:     end for
21:     move cnode to next node in Q;
22: end while
```

Algorithm 1 outlines the workflow of the code analyzer. Its input is the program INFO card. It first (line 1) creates a set *funInfo* with one element for each function in the program, which is called the *INFO card of the function.* It is used as the knowledge base to hold all the information the code analyzer needs in order to recognize seminal behaviors for this function. It initially contains only the function full path_name, its source code and return flag. It then (lines 2-6) breaks the recursive cycles in the call graph and appends the termination conditions to the *funInfo* of the functions where the termination conditions reside. After that, it (lines 7-22) traverses the modified call graph in a postorder (children before parents). For each function, through function *GetAnalyRes* (line 11), it uses the content in *funcInfo* to complete the prompt templates and invokes LLM to get the analysis result for the function. It focuses on finding the seminal behaviors for each function and postpones the mapping to inputs to the end. The algorithm then (lines 12-20) adds the analysis results (after some conversion) of that function into the *funcInfo* of the callers of the function. If a node's callees are all processed, that node is appended (lines 16-17) to the job queue for processing. The process continues until finishing processing the root of the call graph.

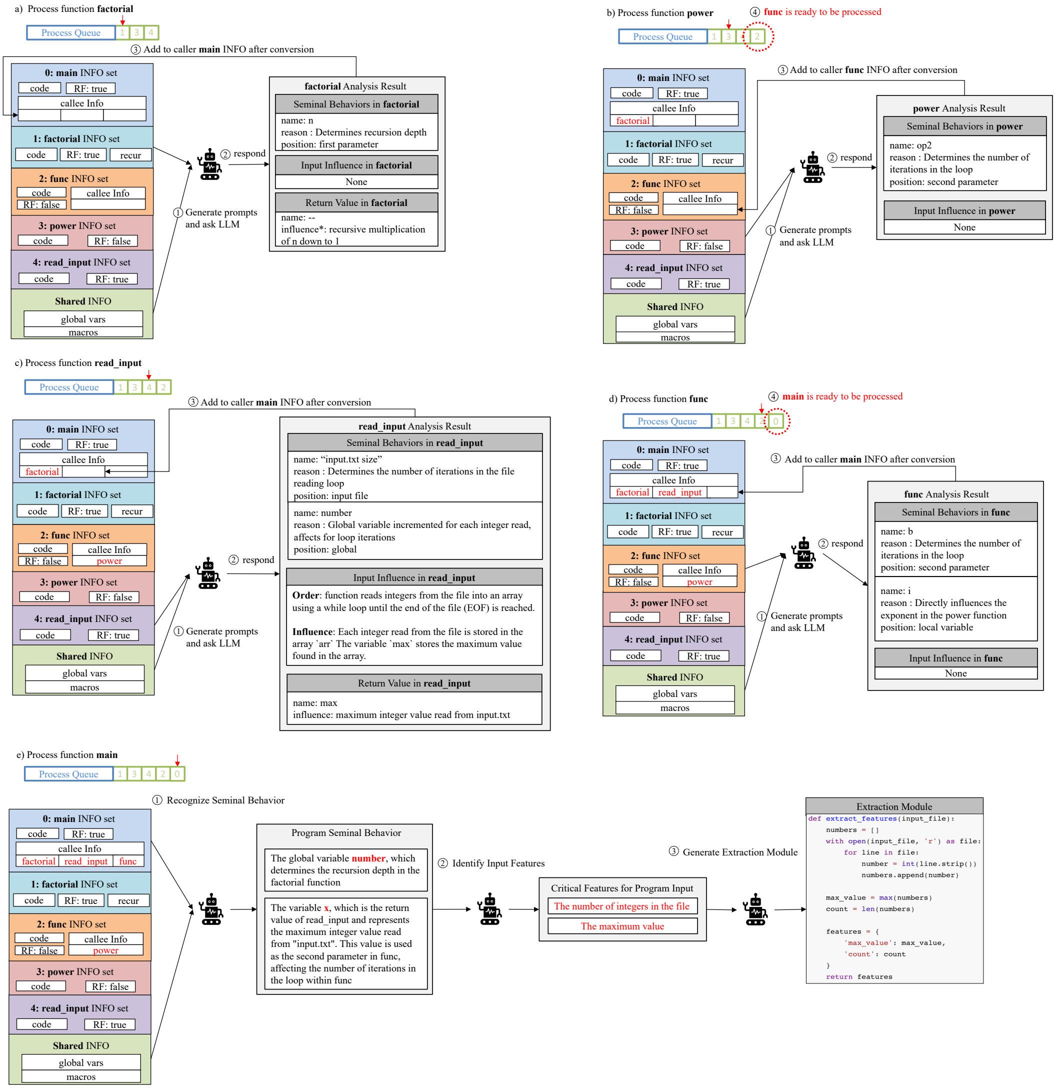
In the algorithm, *GetAnalyRes* works differently for leaf nodes, root node (usually the *main* function), and others which are called non-main caller nodes. Particularly, when it works on the root node, besides recognizing seminal behaviors, it maps the behaviors to the program's inputs (locations or relations), and then calls the *extraction module creator* to create the code for extracting the key features from any given inputs to the program. Note that because program inputs could be read in various points in the program, care must be taken to help LLMs be able to track the order of these reads for it to eventually create a correct feature extraction module. The details are described in Section 4.4.

Figure 5 illustrates the several steps of the iterative process on our example program. Figure 5 (a) shows the initial content of the process queue on top—the three leaf nodes in the post-cycle-elimination call graph—and the INFO cards of all the functions and the shared INFO card (for globals and macros) in the left colorful box. The circled numbers indicate the operations order. The code analyzer collects function "factorial" INFO set, generates prompts and invokes LLM processor. After receiving prompts, LLM returns the analysis result as shown in the right box in Figure 5 (a). The analysis result includes seminar behaviors in the function and how program inputs influence that function and so on. Because function "factorial" is the callee function of the "main" function, this analysis result is added into the "callee Info" area in "main"'s INFO Set with certain conversions or mappings (Section 4.4).

In Figure 5 (b), the analyzer processes function "power" and added the results into the INFO set of its caller "func". Because "func" has no unprocessed callees anymore, it appends "func" to the process queue. Step c) processes function "read_input()" in the same way. Step d) processes function "func". Because "func" calls "power" and "power" analysis results has already been added into its callee area, the processing will use the info of "func" and also the analysis results of "power" when creating the prompts to LLM.

Step e) processes the "main" function. It first recognizes the seminal behaviors based on the info of "main" and the info in its callee area. Note that the recognized seminal behaviors it returns represent the seminal behavior of the entire program, because the analysis results of all other functions have already been propagated to the root either directly or indirectly. For this example, the results from LLMs include two behaviors. One is the value assigned to the global variable "number"; the reason LLMs gives is that it determines the recursion depth and gets value from the while loop in "read_input()". The other is the value assigned to variable "x" in "main" function. The reason it is picked is that as it represents the maximum integer value in program input, it determines the execution time of "func". The algorithm then maps the two behaviors to input features. By analyzing how program input influences the value of "number" and "x", LLMs identify two input features, the number of integers in input and the maximum value among these integers. The final task is to generate a feature extraction module. In the example, this module collects the number of integers in the program input and the maximum value among them.

Because the algorithm applies LLMs to a function each time, it circumvents the token limit of LLMs. And because it uses the call graph to propagate the key info (with size control; see Section 4.4)



name: n
reason : Determines recursion depth
position: first parameter

None

name: --
influence*: recursive multiplication of n down to 1

name: op2
reason : Determines the number of iterations in the loop
position: second parameter

None

name: “input.txt size”
reason : Determines the number of iterations in the file reading loop
position: input file

name: number
reason : Global variable incremented for each integer read, affects for loop iterations
position: global

Order: function reads integers from the file into an array using a while loop until the end of the file (EOF) is reached.

Influence: Each integer read from the file is stored in the array `arr` The variable `max` stores the maximum value found in the array.

name: max
influence: maximum integer value read from input.txt

name: b
reason : Determines the number of iterations in the loop
position: second parameter

name: i
reason : Directly influences the exponent in the power function
position: local variable

None

The global variable **number**, which determines the recursion depth in the factorial function

The variable **x**, which is the return value of read_input and represents the maximum integer value read from "input.txt". This value is used as the second parameter in func, affecting the number of iterations in the loop within func

The number of integers in the file

The maximum value

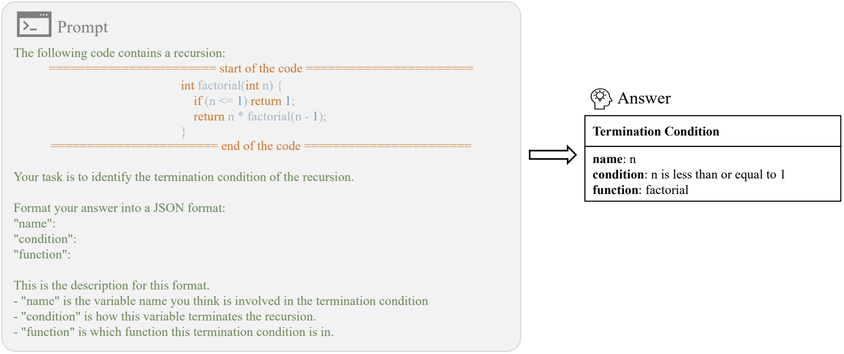
```
def extract_features(input_file):
    numbers = []
    with open(input_file, 'r') as file:
        for line in file:
            number = int(line.strip())
            numbers.append(number)

    max_value = max(numbers)
    count = len(numbers)

    features = {
        'max_value': max_value,
        'count': count
    }
    return features
```

Fig. 5. Illustration of how RACL code analyzer works for each function on the running example in Figure 3 step by step. The left hand of sub-figure a)-e) shows each function's INFO set. They are integrated into prompts. The right hand of sub-figure a)-d) shows analysis result from LLMs. In e), main function is a special case. Analysis results from all the functions are reductive to the main function. We recognize program seminal behaviors, identify input features and generate extraction module in the main function's analysis.

and eventually reduces all the key info into seminal behaviors, it still captures the important effects of program inputs on the whole program.

There are several main questions in the design that are worth further discussions: how are recursions treated, how are the analysis results of a function represented, converted and passed across functions, and how is the mapping to inputs done. We address these questions in the rest of this section.

4.2 Treating Recursions



Prompt

The following code contains a recursion:

```
======================= start of the code =======================
                    int factorial(int n) {
                        if (n <= 1) return 1;
                        return n * factorial(n - 1);
                    }
======================= end of the code =======================
```

Your task is to identify the termination condition of the recursion.

Format your answer into a JSON format:
"name":
"condition":
"function":

This is the description for this format.
- "name" is the variable name you think is involved in the termination condition
- "condition" is how this variable terminates the recursion.
- "function" is which function this termination condition is in.

Answer

Termination Condition

name: n
condition: n is less than or equal to 1
function: factorial

Fig. 6. The instantiated prompt to LLM and LLM's answer to deal with the recursion in the example program.

Recursive cycles would not allow the postorder traversal of the call graph in Algorithm 1 to work as the functions involved in the recursion cycle wouldn't get chance to be processed. Our solution is to break the cycle but have the key info preserved with the help of LLMs. Specifically, it repeatedly removes the edge in a recursive cycle that points to the function that has the lowest index until no recursive calls remain.

But before breaking a cycle, it first uses LLMs to figure out the termination condition of the recursion. Termination conditions are considered as the representatives of the recursion and are included as extra info in the further analysis. If the value of termination conditions are dependent on input features, they will be captured later. To utilize LLMs, we aggregate the source code of all the functions involved in the cycle together and integrate it into the prompt template as shown in Figure 6. LLMs' formatted response is taken as part of the returned value of *EliminateCycle()* (line 2 in Alg. 1), and is added into the INFO card of the function where the termination condition is.

One may worry that the recursion elimination changes the call relations and may cause information loss. For instance, suppose we break the edge from B to A in a A->B->A cycle that has a termination condition in A. When analyzing function B, its INFO card doesn't yet contain the analysis result of A. LLMs will get only some partial results in B along with a comment that "To decide what seminal behaviors are in function A, I need the source code of this function". Those outputs will be converted (detailed in Section 4.4) and added into A's INFO card. When LLMs analyze A, it will automatically leverage the results from B, A's code, and the termination conditions together to do the behavior analysis.

4.3 Results Representation

As mentioned, for scalability, RACL limits the scope of view of LLM to the code of one function each time, plus the analysis results passed to this function from its callees. So for it to work, those analysis results must capture the other functions' essence that is critical to the program's input characterization. The design of what the result should contain is hence important.

In our design, the analysis result on a function includes three parts: seminal behaviors with auxiliary information, program input influence, and return value record. We explain them below; please refer to Figure 5 for examples.

Seminal Behaviors with Auxiliary Information. For a caller function, to fully understand and incorporate the seminal behaviors of its callee functions, we find that it is helpful for the caller to know not only the name of the seminal behaviors of the callee but also how those behaviors influence the overall performance of the callee function. It is hence made as part of the prompts given to LLM, as shown as the 'reason' part in the middle of Figure 7 (a), and the response is made part of the analysis result as illustrated in Figure 5. That information is also helpful when the LLMs needs to drop some less important seminal behaviors (more in Section 4.4).

In addition, to help LLMs later map the seminal behaviors to variables in the callers and eventually to program inputs (more in Section 4.5), it is important to ask LLMs to record the position of each seminal behavior. It is hence made as another part of the prompt template, as shown at the bottom of Figure 7 (a), and the response is made a part of the analysis result as illustrated in Figure 5.

Input Influence. The second part of the recorded analysis result is the influence on this function by program inputs. As shown by the template in Figure 7 (b), LLMs are asked to summarize the order of file reading in the function and how the input (such as which lines) influence variables in this function. This part of results provides extra info to help LLMs later map seminal behaviors to inputs (Section 4.5).

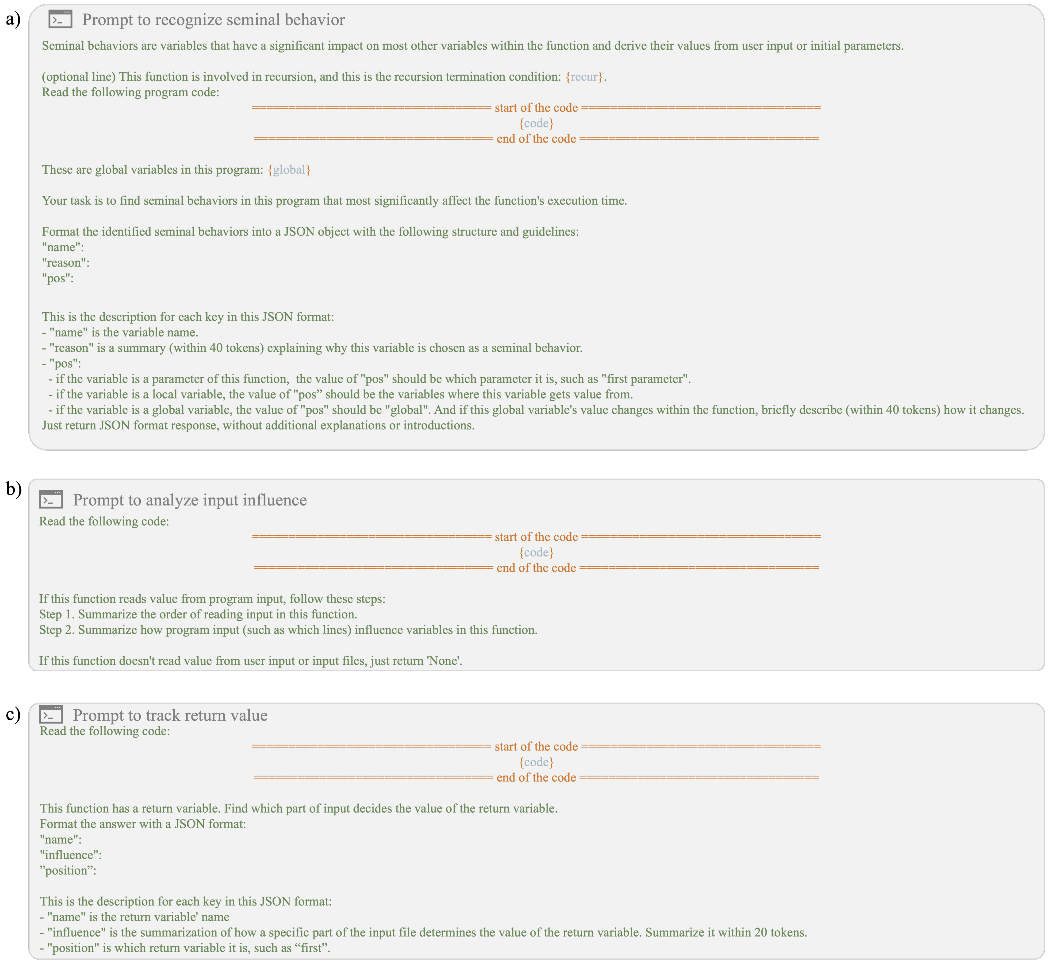
Return Value Record. The third part of the analysis results is the record on the return values of a function. The values in a function can escape to the its caller function through the return statements. As those escaped values can be assigned to seminal behaviors in the caller function, if a function has return variables, the code analyzer needs to invoke LLMs to examine how the return variable gets its values and relays this record to that function's caller as shown by Figure 7 (c). In the record, it includes return variable name, source (how it get its value) and position (which return variable it is). This careful tracking ensures that LLMs can have enough info when doing seminal analysis on the caller function.

4.4 Propagation with Conversion and Size Control

The propagation of the analysis results from callees to a caller is simply done by inserting the analysis results of the callee function into the "callee" field of the INFO card of the caller. But when the analysis moves from callee functions to a caller function, the seminal behaviors chosen in the callee function may shift to variables in the caller function. A typical scenario is that the callee's seminal behavior receives its value from the caller through a parameter passing. In that case, the analysis should update the seminal behavior with the caller's variable. RACL asks LLMs to do variable mapping by including that request in the prompt used for a caller function. LLMs can do that because the analysis result of the callee already contains the information on the relations between the callee's seminal behaviors and the parameters of the function through a `position` field.

The analysis result of a function could grow larger and larger as RACL goes up along the call graph; the context passed to the LLMs would grow at an even faster rate for the aggregation effects, and eventually go beyond the limit of LLMs. But in reality, that kind of explosion will not happen. LLMs have a default limit on the maximal length the generated text can be at one request. As a result, the analysis result a function passes to its callers cannot exceed a limit, no matter how close that function is to the root of the call graph.

That limit forces LLMs to be selective in reporting seminal behaviors. RACL does not explicitly rank seminal behaviors. It relies on the internal of LLMs to automatically filter and prioritize seminal behaviors during the analysis process. To help LLMs in that process, we embed some generic instructions in the prompts. One of those instructs LLMs that if a seminal behavior in a callee function corresponds to a function parameter, LLM should replace its reference with the corresponding variable in the caller function. This ensures that behaviors are propagated meaningfully through the call graph rather than redundantly repeated. The others are instructions in the prompt that ask the LLM to explain why a behavior is chosen as seminal by providing a



a)

```
Prompt to recognize seminal behavior
Seminal behaviors are variables that have a significant impact on most other variables within the function and derive their values from user input or initial parameters.

(optional line) This function is involved in recursion, and this is the recursion termination condition: {recur}.
Read the following program code:
                    ================================ start of the code ================================
                                                      {code}
                    ================================ end of the code ================================

These are global variables in this program: {global}

Your task is to find seminal behaviors in this program that most significantly affect the function's execution time.

Format the identified seminal behaviors into a JSON object with the following structure and guidelines:
"name":
"reason":
"pos":


This is the description for each key in this JSON format:
- "name" is the variable name.
- "reason" is a summary (within 40 tokens) explaining why this variable is chosen as a seminal behavior.
- "pos":
  - if the variable is a parameter of this function,  the value of "pos" should be which parameter it is, such as "first parameter".
  - if the variable is a local variable, the value of "pos” should be the variables where this variable gets value from.
  - if the variable is a global variable, the value of "pos" should be "global". And if this global variable's value changes within the function, briefly describe (within 40 tokens) how it changes.
Just return JSON format response, without additional explanations or introductions.
```

b)

```
Prompt to analyze input influence
Read the following code:
                    ================================ start of the code ================================
                                                      {code}
                    ================================ end of the code ================================

If this function reads value from program input, follow these steps:
Step 1. Summarize the order of reading input in this function.
Step 2. Summarize how program input (such as which lines) influence variables in this function.

If this function doesn't read value from user input or input files, just return 'None'.
```

c)

```
Prompt to track return value
Read the following code:
                    ================================ start of the code ================================
                                                      {code}
                    ================================ end of the code ================================

This function has a return variable. Find which part of input decides the value of the return variable.
Format the answer with a JSON format:
"name":
"influence":
”position”:

This is the description for each key in this JSON format:
- "name" is the return variable' name
- "influence" is the summarization of how a specific part of the input file determines the value of the return variable. Summarize it within 20 tokens.
- "position" is which return variable it is, such as “first”.
```

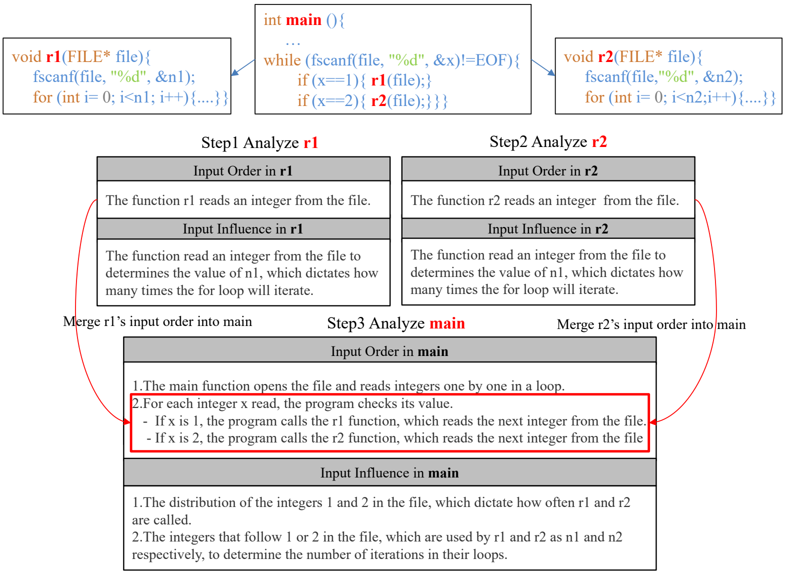
Fig. 7. Prompt templates used for leaf functions. The templates are instantiated by the RACL code analyzer by filling the braces{} with the information extracted from the INFO cards.

justification in the "reason" field of the response. This explanation mechanism enforces internal consistency in LLM's reasoning and encourages it to prioritize behaviors that are more fundamental to program performance rather than just have some surface-level correlations.

4.5 Mapping to Inputs and Extraction Module Generation

Mapping to Inputs When the analysis reaches the root of the call graph (i.e., the "main" function), it does the seminal behavior analysis as usual, and then maps the seminal behaviors to the content or features of the program inputs. Without extra info, by examining just the code of the "main" function, LLMs would not be able to do that: A program can possibly read input files in any of its functions. LLMs can achieve that in RACL is because the RACL has carefully kept the necessary info—the "input influence", "position" of seminal behaviors, and the "return value record"—in the INFO cards, which are propagated through the call graph.

To help readers see the input order complexity and how the info tracking in RACL helps, we draw on a simple example in Figure 8. Based on algorithm.1, the code analyzer starts with functions



```
void r1(FILE* file){
    fscanf(file, "%d", &n1);
    for (int i= 0; i<n1; i++){....}}
```

```
int main (){
    ...
while (fscanf(file, "%d", &x)!=EOF){
    if (x==1){ r1(file);}
    if (x==2){ r2(file);}}}
```

```
void r2(FILE* file){
    fscanf(file,"%d", &n2);
    for (int i= 0; i<n2;i++){....}}
```

Step1 Analyze **r1**

Input Order in **r1**
The function r1 reads an integer from the file.
Input Influence in r1
The function read an integer from the file to determines the value of n1, which dictates how many times the for loop will iterate.

Step2 Analyze **r2**

Input Order in **r2**
The function r2 reads an integer from the file.
Input Influence in r2
The function read an integer from the file to determines the value of n1, which dictates how many times the for loop will iterate.

Merge r1's input order into main
Merge r2's input order into main

Step3 Analyze **main**

Input Order in **main**

1.The main function opens the file and reads integers one by one in a loop.
2.For each integer x read, the program checks its value.
- If x is 1, the program calls the r1 function, which reads the next integer from the file.
- If x is 2, the program calls the r2 function, which reads the next integer from the file

Input Influence in **main**

1.The distribution of the integers 1 and 2 in the file, which dictate how often r1 and r2 are called.
2.The integers that follow 1 or 2 in the file, which are used by r1 and r2 as n1 and n2 respectively, to determine the number of iterations in their loops.

Fig. 8. An example to illustrate the merge process for program inputs. The top directed graph is a simple call graph with three functions. The bottom graph shows the code analyzer merges input order and input influence based on function call. At the end of the process, after analyzing main function, the code analyzer understands how the program read contents from inputs.

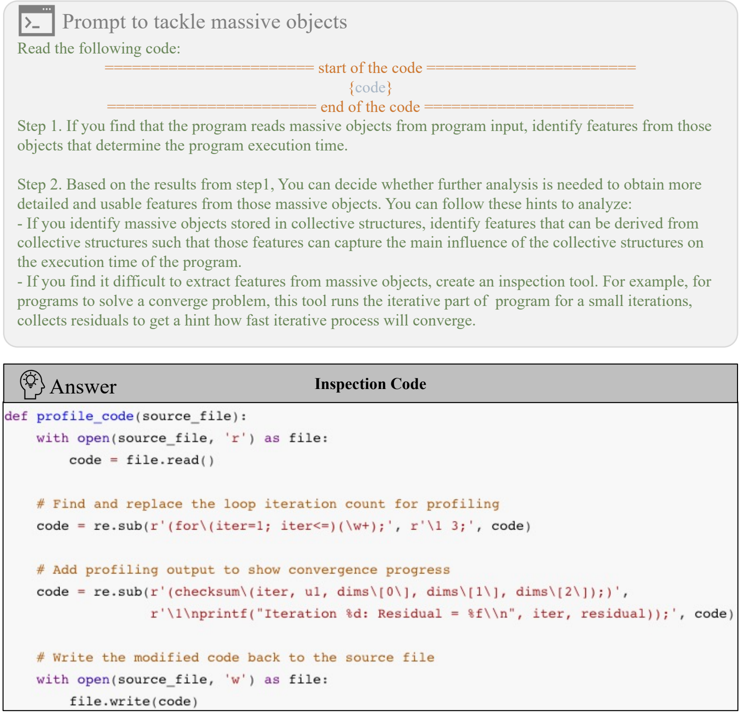
"r1" and "r2". The analysis results of "r1" and "r2", including *input order* and *input influence*, are propagated to "main". The prompts used for "main" function require LLMs to merge the results from "r1" and "r2", and summarize the input order and influence for the entire program as the result in Figure 8 shows. This example also shows in case where the reading order of inputs depend on certain conditions on some value read earlier from the input, the result from LLMs will include those conditions. In the input order of the “main" function, LLMs summarize conditions to call the function “r1" or “r2".

It is worth noting that the mapping result is not always certain elements in the inputs. It sometimes is just descriptions of certain input features. As the bottom row in Figure 5 shows, the mapping result of our running example is two critical features of the input described in text, "the number of integers in the file" and "the maximum value in the file".

Generating Feature Extraction Module The final job of RACL code analyser is to generate a feature extraction module. It can be used before the target program starts. Whenever a new input comes in, it can quickly extract the necessary input features and apply a predictive model to predict the program's behaviors and hence guide runtime optimizations. Because the previous step already produces the descriptions of the key input features, the generator of the module just needs to ask LLM to do the code generation. That is what the first phase of our module generator does.

But some input features identified by RACL analyzer are massive objects in an input on which LLM has a hard time to extract some concise features. An example is a numerical solver, the number of iterations for it to converge (and hence its running time) depends on the values of the input grid which can consist of many points. Although the characterization is theoretically correct, given the vast numbers of values in such objects, it is impractical to treat each one as an individual input feature for practical usage.

To address this issue, we incorporate the concept of *inspection* into our module generator. As shown in Figure 9, we provide a dedicated prompt template to guide the LLMs in deciding whether to create an inspection tool. Step 1 of the prompt explicitly asks the LLM to identify cases where a program reads massive objects from input and struggles to extract concise features. Step 2 then provides guidelines on how the LLM should analyze these objects, including strategies such as identifying collective structure properties or, when necessary, creating an inspection tool (the last bullet in Step 2). In the prompt, we also describe a scenario in which an inspection tool is useful—such as in convergence-based computations, where the program iteratively refines a solution over multiple steps. The generation of inspectors is an integral part of the overall workflow and is used during input feature recognition and extraction module generation.



Prompt to tackle massive objects

Read the following code:

======================= start of the code =======================

{code}

======================= end of the code =======================

Step 1. If you find that the program reads massive objects from program input, identify features from those objects that determine the program execution time.

Step 2. Based on the results from step1, You can decide whether further analysis is needed to obtain more detailed and usable features from those massive objects. You can follow these hints to analyze:
- If you identify massive objects stored in collective structures, identify features that can be derived from collective structures such that those features can capture the main influence of the collective structures on the execution time of the program.
- If you find it difficult to extract features from massive objects, create an inspection tool. For example, for programs to solve a converge problem, this tool runs the iterative part of program for a small iterations, collects residuals to get a hint how fast iterative process will converge.

Answer

Inspection Code

```
def profile_code(source_file):
    with open(source_file, 'r') as file:
        code = file.read()

    # Find and replace the loop iteration count for profiling
    code = re.sub(r'(for\(iter=1; iter<=)(\w+);', r'\1 3;', code)

    # Add profiling output to show convergence progress
    code = re.sub(r'(checksum\(iter, u1, dims\[0\], dims\[1\], dims\[2\]);)',
                  r'\1\nprintf("Iteration %d: Residual = %f\\n", iter, residual));', code)

    # Write the modified code back to the source file
    with open(source_file, 'w') as file:
        file.write(code)
```

Fig. 9. Prompt template for creating the inspection tool and an example inspection tool produced by LLMs.

5 Predictive Model Construction

To use the identified input features in input-centric optimization, a predictive model is built to predict the behaviors of the program from the extracted input features. The prediction can then be used for guiding runtime optimizations. Building such a model is not part of MERIC. Many classification and regression methods are applicable to input-behavior modeling. In our evaluation of the usefulness of the identified features (Section 6), we use XGBoost (Extreme Gradient Boosting) [2] as the primary approach because it is lightweight and has been proved effective on a broad range of predictive tasks [42]. Other machine learning methods could also be used.

XGBoost is a scalable and efficient machine learning algorithm based on gradient boosting decision trees. It builds an ensemble of weak learners—typically decision trees—by sequentially adding models that correct the residual errors of prior models. In our work, we employ XGBoost to model the relationship between identified input features and runtime behaviors. Given a training dataset consisting of input features $\mathbf{x}_i \in \mathbb{R}^m$ and corresponding target labels y_i, XGBoost aims to learn a function $\hat{y}_i = \hat{y}(\mathbf{x}_i)$ that accurately predicts y_i. The targets y_i represent either the program's runtime (for regression tasks) or the optimal OpenMP scheduling configuration (for classification tasks).

XGBoost trains the model by minimizing a regularized objective function of the form:

$$\mathcal{L}(\theta) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \sum_{k=1}^{K} \Omega(f_k),$$

where $\ell(y_i, \hat{y}_i)$ is the loss function measuring the difference between the predicted and true values, and $\Omega(f_k)$ is a regularization term penalizing the complexity of the k-th decision tree f_k.

The regularization term $\Omega(f_k)$ is typically defined as:

$$\Omega(f_k) = \gamma T_k + \frac{1}{2}\lambda \sum_{j=1}^{T_k} w_{kj}^2,$$

where T_k is the number of leaves in tree f_k, w_{kj} is the weight of the j-th leaf in the k-th tree, and γ, λ are regularization parameters that control model complexity and overfitting.

XGBoost automatically constructs the models by iteratively adding trees f_k to minimize $\mathcal{L}(\theta)$. The final prediction is the sum of outputs from all trees:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

where $\mathcal{F}$ denotes the space of all candidate regression or classification trees.

In our experiments, we employed the standard five-fold cross-validation approach, where in each iteration, the dataset is randomly split into an 80/20 train-test ratio. The reported results reflect the average accuracy on the test dataset across all folds.

6 Evaluation

We implement MERIC with GPT-4 APIs without fine-tuning. To evaluate the efficacy of MERIC, besides examining the accuracy in running time prediction, we test three input-centric optimizations based on the inputs features characterized by MERIC on eight programs.

Use Case 1: OpenMP Parallelism As a popular parallel programming models, OpenMP [4] automatically parallelizes a code region based on the directives. The optimization we focus on is the parallel settings, that is, to determine the best parallel scheduling policies to use (static, dynamic, or guided) and the best numbers of threads to use (8,12,16). We consider three scheduling polices, static (assigning chunks of iterations to each thread at compile time), dynamic (assigning chunks of iterations to threads at runtime), and guided (similar to dynamic but with decreasing chunk sizes). Together they provide 9 possible combinations. For a given program, the best parallel setting often differs for different inputs. We try to use the identified key input features as inputs to a XGBoost-based predictive model so that it can predict the best parallel setting for a given input to a program.

Use Case 2: Shortest Job First (SJF) Schedule The second use is to guide SJF schedule. SJF is one of the basic job scheduling algorithms used in operating systems. When a number of jobs need to be served in a resource-constraint environment, SJF prioritizes jobs taking the least amount of time. It guarantees to minimize the average job wait time. The challenge for using SJF is that it needs to know the duration of each job beforehand, which can be addressed by a predictive model on the running time of a program based on its input features.

Use Case 3: Serverless Computing The simplicity of SJF scheduling gives conveniences in fully analyzing the results, but it falls short in showing how useful MERIC can be for real-world uses. We hence experiment a third use case. It is to guide runtime scheduling in a full-fledged serverless computing platform, OpenWhisk [26, 27]. Serverless computing is an influential cloud computing paradigm, known for its free of provisioning needs and appealing cost effectiveness. Its controller schedules arriving jobs to the many workers in a datacenter. The scheduling algorithm is complicated, considering the availability of resources in the datacenter, warm or cold starts, and other factors. In our experiment, we use the scheduler published recently by Hui and others [13]. Its scheduling algorithm achieves the state-of-the-art performance by considering those mentioned factors but also the estimated job length. Their study however didn't consider the impact to job lengths from inputs, and simply uses the average length of a serverless function. In our experiment, we examine the performance improvement when we replace the average length with the input-based predicted length.

6.1 Methodology

Comparison Counterparts We compare MERIC with two alternatives. The first is the prior solution in current input-centric optimizations, a profiling-based method (represented as "Profiling") [14]. As we have mentioned in Section 1, it conducts many detailed slow profiling runs to collect the counts of the calling frequencies of each function, the tripcounts of every loop, and many other values in the program executions on various inputs. It uses statistical correlation analysis to then identify seminal behaviors and construct predictive models. The model features include interface behaviors (e.g., behaviors directly attained from inputs) and internal key behaviors (loop trip counts, function call frequency, etc). This method is tedious and time-consuming.

The second is the *map-reduce* method, the typical approach in LLM community (e.g., langchain [16]) to dealing with large documents. This method decomposes a large document into smaller segments for analysis. In the "map" phase, each segment is independently processed to generate a summary or response. Subsequently, in the "reduce" phase, these discrete outcomes are combined to form a comprehensive summary or answer for the entire document. The divide-and-conquer idea aligns with MERIC but it is oblivious to code structure. Its approach to the propagation and consolidation of results is simple.

Benchmarks We use ten programs for our evaluations. They include four real-world applications with thousands to near a hundred thousands of lines of source code and tens to hundreds of functions. Their sizes all exceed the token limit of the LLMs (GPT4), making them suitable for testing MERIC, the scalable solution proposed in this work. Because the use case 1 in our experiment is about OpenMP parallelism, we in addition include the six programs in NAS Parallel Benchmarks (NPB), an OpenMP benchmark widely used for OpenMP performance studies [1, 7, 20, 23]. These programs have modest sizes; they are for assessing the effectiveness of the identified input features rather than scalability of the solutions. Table 1 lists all the ten programs, including their source code size, number of functions, inputs and their sizes, and the input features identified by MERIC.

We collect 25–256 inputs for each of the programs. During the collection, we try to ensure that the inputs are typical in the normal executions of the benchmarks. For example, the original input for CG program just defines the sparse matrix size, how many nonzero values in each line, number of iterations and shift value. Based on these hyper-parameters, the original CG program randomly generate a simple sparse matrix. We collect real-world sparse matrices in SuiteSparse Matrix Collection [5] and transform them into the appropriate format which CG program can directly use.

Table 1. Benchmarks and Input Features by MERIC. (Underlined features are those identified by *map-reduce* method.)

Program	Program Description	LOC	# Func	#Inputs	Input File Size	Input Features by MERIC
CG [24]	Conjugate Gradient	694	4	200	157KB~990MB	NA,NONZERO,SHIFT values from inspector (rnorm1,rnorm2,rnorm3)
MG[24]	Multi-Grid on a sequence of meshes	1032	11	56	12KB~1.1G	Level, Three Dimensions, Iterations
FT[24]	Discrete 3D fast Fourier Transform	967	18	256	48KB~2.6G	Three Dimensions, Iterations
BT[24]	Block Tri-diagonal solver	2963	19	138	72KB ~1.6G	Iterations, Dt, Three Dimensions
SP[24]	Scalar Penta-diagonal solver	2781	20	144	72KB ~1.6G	Iterations, Dt, Three Dimensions
LU[24]	Lower-Upper Gauss-Seidel solver	3475	16	140	72KB ~1.6G	Iterations, Dt, Three Dimensions
MCF[19]	Single-depot vehicle scheduling in public mass transportation	3740	37	52	126KB ~788MB	number of timetabled trips number of dead-head trips
LBM[28]	Lattice Boltzmann Method	1350	21	180	–	grid dimensions time steps simulation setup
x264[37]	A free application for encoding video streams into the H.264/MPEG-4 AVC format	96K	620	90	131MB - 4.35GB	Total Frames Resolution, Bitrate, Quality
parser[6]	A syntactic parser of English	11K	456	25	282B - 703KB	the number of words in the files the complexity of the words average length of the sentences dictionary size

Machines We use a local Linux machine to collect execution time. And the hardware used in serverless computing experiment is a 64-node cluster. The details of the local machine and the machine in the cluster are as follows:

	Cluster Node	Local Machine
CPU Model	AMD EPYC 7302P 16-Core	Intel(R) Xeon(R) CPU E5-2630 40-Core
CPU GHz	1.5	1.2
Memory	128GB	512GB
System	Ubuntu 20.04.6 LTS, g++ 9.4.0	Ubuntu 20.04.6 LTS, g++ 9.4.0

6.2 Identified Input Features and Time Comparison

The rightmost column in Table 1 shows the input features identified by MERIC. For all the programs, the features by the map-reduce method all include "every element in the input file" as a feature and some made-up features. We hence added a filtering step to remove those useless features. The remaining features are subsets of those identified by MERIC, marked with underlines in the rightmost column of Table 1.

Unlike MERIC and *map-reduce* methods, the profiling-based method may include some key behaviors inside a program (e.g., the tripcounts of some loops) as part of the seminal behaviors. Because those behaviors remain unknown until some portion of the program execution has passed, using the seminal behaviors by that method for input-centric optimization may suffer delays. For that reason, the original profiling-based method [14] allows the use of a threshold (called *earliness*) for picking the seminal behaviors. In our experiments, we include two settings for that method: Profiling-1 includes no internal behaviors but only interface values, and profiling-2 includes the key internal behaviors that appear in the first 20% portion of an execution (i.e., *earliness=80%*). Like

MERIC and *map-reduce* methods, profiling-1 is not subject to delays in guiding runtime predictions and optimizations, while profiling-2 is subject to 20% delays.

Tables 3,4 report the results. For most programs, the input features identified by MERIC match with interface variables used by the profiling-based method. And for Profiling-2 method, we observe that most seminal behaviors are statistically related to interface variables. Hence, as shown in Table 3, MERIC, Profiling-1 and Profiling-2 reach the same accuracy for 6 programs. The differences are on CG, MCF, x264 and parser. For CG, beside three input features all the methods find, our MERIC method identifies three extra features via the inspector. Profiling-based methods cannot get those features, but profiling-2 identifies an additional loop trip count. For MCF, x264 and parser, we try to select the similar number of features as reported in previous method [14]. Profiling-1 method selects the same interface variables as the input features identified by MERIC for MCF and x264, but it only considers "the number of words" as one interface variable for parser. This is because other input features identified by MERIC for parser are not directly interfaced by the program, but are analyzed based on program inputs. Profiling-2 method adds six, one and three extra loop trip counts for MCF, x264 and parser, respectively, as seminal behaviors. However, the profiling methods don't achieve better accuracy than MERIC except using Profiling-2 for MCF.

The times taken by the methods differ substantially. The profiling-based method requires detailed code instrumentation and profiling of many runs of the program on many different inputs and then uses machine learning to do correlation analysis to find out the critical features. The profiling needs to record the trip counts of all loops, the frequencies of all function calls, and so on. Each profiling run is 2-8X longer than the native run, and there are hundreds of runs to profile for just one program. The process is time consuming and adds lots of burden to programmers. As shown in Table 2, it needs on average 9.6 hours for each program, even without counting the extra time needed for the correlation analysis. MERIC (via ChatGPT-v4) just needs on average 13 minutes (a 44× reduction). Moreover, if the LLMs are deployed locally, the time can be reduced even further: The "MERIC (local LLM)" row in Table 2 shows that the time estimated with Llama70B (since ChatGPT-v4 is not publically available for local deployment) on a machine with four A100 GPUs [36]. On average, only 77 seconds (a 450× reduction) are needed per program. The large time reduction is significant, as it shows that it is possible to integrate the agile automatic program input characterization into the workflow of program compilation, hence potentially removing the long-standing barrier for practical adoption of input-centric code optimizations.

Table 2. Comparison of Time Needed by Input Characterization. ("MERIC (local LLM)" are estimated times with Llama70B on 4xA100 GPUs.)

	CG	MG	FT	BT	SP	LU	MCF	LBM	x264	parser	Average
Profiling method	3.4h	5.6h	10.1h	3.3h	4.8h	6.4h	3.3h	8.8h	44h	6.6h	9.6h
MERIC (remote LLM)	157s	314s	457s	551s	517s	444s	959s	474s	2481s	1326s	768s
MERIC (local LLM)	14s	31s	45s	74s	62s	63s	97s	44s	223s	117s	77s

6.3 Running Time Prediction

To see the usefulness of the identified input features, based on the input features identified by each of the three methods (MERIC and the two baseline methods), we built three predictive models for predicting the running times of the runs of each program. The models have the same form, but differing in the input features they use. We show the mean absolute percentage errors (MAPE) of the predicted running time for each program in Table 3. For most programs, the error rates of MERIC hover around 0.1, ranging from the lowest at 0.056 for parser to the highest at 0.298 for MG.

Table 3. MAPE for Execution Time Prediction

	CG	MG	FT	BT	SP	LU	MCF	LBM	x264	parser
MERIC	0.073	0.298	0.179	0.064	0.106	0.122	0.093	0.102	0.123	0.056
Map-Reduce	0.203	0.334	0.179	0.064	0.106	0.122	0.093	0.573	0.152	0.131
Profiling-1	0.203	0.298	0.179	0.064	0.106	0.122	0.093	0.102	0.123	0.4
Profiling-2	0.186	0.298	0.179	0.064	0.106	0.122	0.084	0.102	0.124	0.157

Table 4. Prediction Accuracy for Configuration Selection

	CG	MG	FT	BT	SP	LU	MCF	LBM
MERIC	95.8%	89.3%	93.4%	91.6%	91.7%	97.5%	86.4%	95%
Map-Reduce	85.3%	71.4%	93.4%	91.6%	91.7%	97.5%	86.4%	55.6%
Profiling-1	85.3%	89.3%	93.4%	91.6%	91.7%	97.5%	86.4%	95%
Profiling-2	91.5%	89.3%	93.4%	91.6%	91.7%	97.5%	88.5%	95%

The accuracy is similar to that from the heavy profiling-based method. In comparison, the error rates of the map-reduce method are significantly higher.

6.4 OpenMP Parallelism Optimization

We show the prediction accuracy of the best parallel settings in Table 4. (x264 and parser are not written in OpenMP.) The predictors built on the features from MERIC show high prediction accuracy, from the lowest 86.4% for MCF to the highest accuracy 97.5% for LU. The results by MERIC are similar or even better (thanks to the inspection-based features) than those from the heavy profiling-based method. The accuracy by the map-reduce method is significantly lower. Figure 10 shows the speedups that the predicted parallel settings achieve. The baseline performance of a program is the best performance it can achieve with a single parallel setting, that is, the best performance it can achieve in the input-oblivious way. We use boxplots to show the speedups by MERIC, map-reduce, and the optimal settings (which is attained by running each in all settings and pick the best). Among the eight programs, four show clear input-sensitivity in terms of the best parallel setting. MERIC brings near optimal speedups in all of them, significantly higher than those from the map-reduce method.

6.5 SJF Scheduling

In the SJF experiment, the predictive model is built to predict the running time of each program on a given input, and the prediction is used in SJF scheduling. Figure 11 shows the total job wait times when SJF uses the actual running times, the times predicted based on MERIC, and the average times of a program (input-oblivious case). The number of jobs ranges from 50 to 450, with each being one invocation of one of the eight programs on a randomly chosen input with k threads, where k is chosen randomly from the set {1,4,8,12,16}. We tested it on three settings, with 16, 32, and 40 cores. The predicted times by MERIC can let the SJF schedule produce near optimal wait time, which is much shorter than the input-oblivious case.

6.6 Serverless Computing

In this experiment, the optimized controller [13] in OpenWisk schedules jobs by leveraging the estimated job running times (the same kinds of predictive models as in Section 6.3 are used). We create a serverless function for each of the eight programs. The workloads in the experiments are a

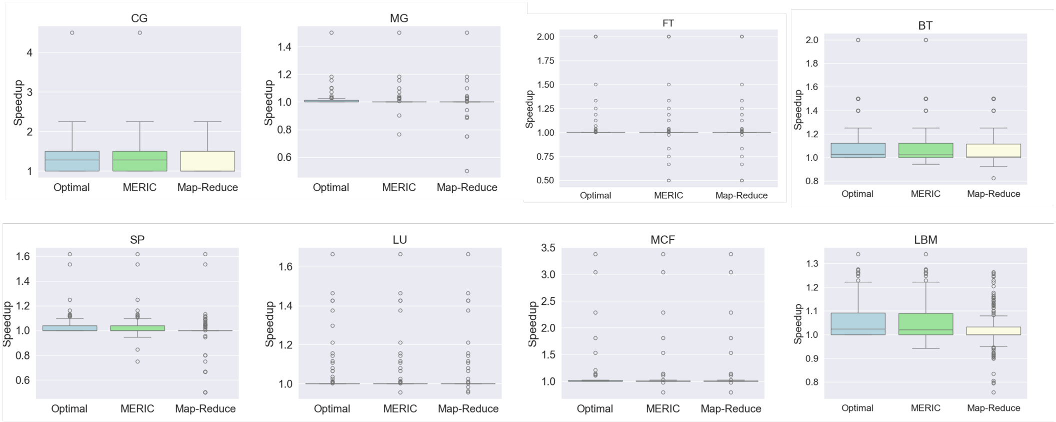


Fig. 10. Speedups brought by OpenMP parallelism optimizations on program CG, SP, BT, and LBM. The other four programs are not sensitive to parallelism optimizations and show similar insignificant speedups in all methods.

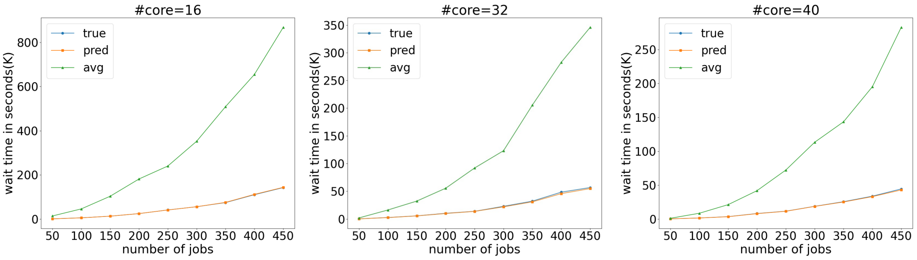


Fig. 11. Total wait time in SJF schedules. The 'true' curves and 'pred' curves are largely overlapped.

series of 800 calls to those functions with each call taking a randomly chosen input. We examined the traces published by Azure [30] and derived two situations respectively with **bursty** and **steady** workloads. In the **bursty** workload, the arrival interval for the functions is in [1–1.68ms]. In the **steady** workload, the function is randomly picked to get invoked in ranges [50–84ms].

The goal of serverless scheduling is to make the jobs meet their required latency (called Service Level Objective (SLO) latency) while minimizing the resource usage (in terms of # of vCPUs). We experiment with two SLO latency levels. Let t be the time needed by a function to complete when it runs alone on one vCPU. SLO latency level is defined as the ratio between the SLO latency and t. An SLO latency level of 0.8x refers to the case where the acceptable maximal latency is 0.8 times t. The two SLO levels we experiment with are 0.8x (**moderate**) and 1.0x (**relaxed**).

Figure 12 shows the results. SLO hit rate is the percentage of jobs that meet the SLO latency requirement. Due to the space limit, we show only the result on the steady workload. The busty workload shows a similar trend. Figure 12 (a) and (c) show the SLO hit rate for the moderate SLO and relaxed SLO, respectively, while (b) and (d) show the corresponding resource usage. In addition to the default results (using the average time of a program) and our results (using the input-based predicted time), we also include the results of using the actual time (called "Oracle"). By enabling input-sensitive scheduling, MERIC helps the serverless computing achieve 33% and 16% higher average SLO hit rates in the two SLO levels, and at the same time, reduces the resource usage by 65% for both. The results are close to the "Oracle".

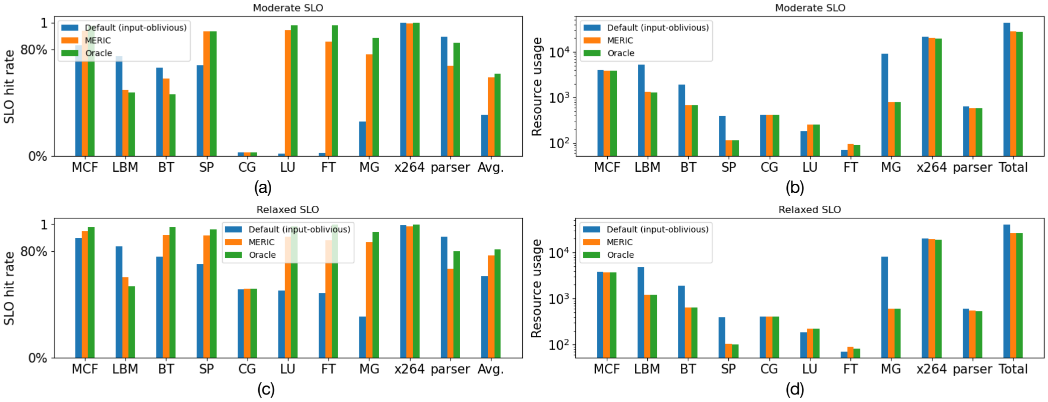


Fig. 12. The performance and resource usage in serverless computing when the workload is steady. (a) and (c) show the SLO hit rate in moderate and relaxed SLO level, respectively. The higher, the better. (b) and (d) show the resource usage. The lower, the better.

We dive deeper into program MG to see some insights behind. On the program, the "Default" case allocates more resources but sees significantly lower SLO hit rates. It is primarily due to misleading effects of its use of average time of a program. While the average execution time of MG decreases with increased #vCPUs, it is not the case for all its executions (due to the time waiting for resources, and communication overhead) except those on very large inputs. Because those small portion of runs have longest time, the average time still shows that trend. Unaware of the input-sensitivity, the controller allocates excessive resources to MG and suffers poor performance.

6.7 Overhead

One of the factors to consider for input-centric optimizations is the runtime overhead of getting the input features and making the predictions. There are various ways to minimize or hide the overhead. We can compute the values of all the features at program startup through extraction modules. These extraction functions run as parallel processes alongside the main execution, allowing the feature values to be obtained asynchronously. Once their values are obtained, the results can be used by runtime for predictions. Since all the features are extracted in the background, the overhead will be hidden.

Because the purpose of the use cases in the previous sections is assessing the quality of the identified input features, we did not give special treatment to the overhead and excluded the overhead from the measured performance. To demonstrate that the overhead is negligible in our use cases, we quantify it by comparing it to the total program execution time. As described in Table 1, most of the identified input features are several numbers lying at the beginning of the input files, and our predictor is based on XGBoost which is lightweight. Our measurement shows that the overall overhead on an execution is all below **0.12%** for the program executions. Even though some input features require more than constant time to extract, our empirical measurements indicate that the associated overhead remains minimal in practice. For example, we measured the overhead on `parser`, which uses "average sentence length" as a key feature. Our results show that the overall overhead remains below **0.05%** of the program's execution time.

7 Discussion

Validation Concerns. A risk of using machine learning methods—especially LLMs—to program analysis is that it may not be always reliable, particularly when outputs are represented in natural language rather than quantitative values. It is however less of a concern for MERIC. Input-centric optimizations are about selecting one optimization decision from a set of candidate decisions. It is important to note that all of the candidate decisions in the target input-centric optimization (e.g., tile size, number of threads to use, task schedules) are designed to be legal and sound; they just lead to different performances. Moreover, in this context, LLMs do not directly change the original program, but identify key input features; the transformations to the original programs are still done by compilers or runtime. They leverage the results from LLMs as hints. If the hints are wrong, the optimizations may not be very effective in delivering the best running speed, but would not cause correctness issues.

MERIC is designed as a fully automatic framework–no manual validation is in its workflow. The quality of the input features identified by MERIC is eventually validated by their effectiveness in supporting the predictive models in making accurate predictions. Adding automated consistency checks for LLMs' intermediate results such as info card generation (and using them for auto-refinement) might further enhance the confidence of the result. It is hard for such tools to offer complete checks due to the hard-to-resolve aliases and other code complexities, but some simpler options do exist. An example is a compilation pass that verifies whether identified seminal behaviors satisfy their definition. For example, If a local variable is incorrectly marked as a seminal behavior but does not receive its value from program inputs or function parameters, the checker could filter it out, as it does not meet the criteria. We experiment with adding such a checker into the MERIC workflow at the step where the seminal behaviors of a function are identified. The error rate for the analysis of a function is defined as the ratio of invalid seminal behaviors to the total recognized seminal behaviors. The average error rate detected by the checker is **14.4%**, ranging from **9.7%** on `MCF` to **21.2%** on `FT`. Moreover, we observed that nearly **90%** of the invalid seminal behaviors of a function were dropped automatically by the original MERIC (without the checker) during a propagation from a callee to its caller function, as those invalid behaviors typically lack corresponding matches in the caller functions. (Recall as we discussed in Section 4.4, we embed some instructions in the prompts to help LLMs automatically filter and prioritize seminal behaviors during the propagations.) As a result, when the propagation reaches the main function, almost no influence from such invalid behaviors remain, and hence causes little if any effect on the final analysis results, in terms of the identified key input features of the programs and the prediction accuracies of the predictive models.

Indirect Function Calls. We use call graphs generated by Doxygen. Doxygen has internal ways to treat indirect calls so we didn't give special treatments. There could be some indirect calls not captured by doxygen but the influence on the experiments is not significant as reflected by the high accuracies observed in the evaluation. Finding the best ways to treat indirect calls in call-graph construction is still an active research topic. It is orthogonal to our work: Any future improvement of it can directly help unleash the potential of our method.

Generated Extraction Module. As we introduce in Section 4.5, the main task of RACL is to recognize critical input features. To facilitate these input features, we design a subsequent task to generate extraction modules. Currently we rely on the ability of LLM itself for code generation, which works fine for our experimented programs. If a user would like to further ensure the quality of the generated feature extraction code, he could optionally examine the code or the extracted features manually as both the code and the features are human-readable.

Usage and Challenge of MERIC. With a set of APIs, MERIC can serve as an end-to-end tool accessible for broader uses. Through the APIs, users can provide the source code of their application, and MERIC will analyze the code and return the recognized input features that influence program execution, along with a feature extraction module that enables automatic extraction of these features at runtime. Users can then use these extracted features to build their own prediction models for adaptive optimizations. Additionally, if users can provide sample input datasets, MERIC can further assist in training predictive models based on the identified input features and generating optimization suggestions for specific uses.

While MERIC is designed to be general, there are certain scenarios where it may face challenges. MERIC requires access to the source code of the application of interest. It is okay if the application include the use of some common binary libraries. But if a large chunk of the application's source code is not accessible to MERIC, MERIC will have a hard time analyzing it since LLMs have no enough info to do its code analysis.

Locally Deployed LLMs versus LLMs on Remote. Our primary goal in this work is to demonstrate that LLMs have the capability to solve the input characterization problem effectively. As shown in Table 2, we highlight that local deployment of LLMs, if feasible, can significantly speed up analysis time by eliminating API latency. Recent open-source locally deployable models (e.g., DeepSeek[12]) are achieving comparable performance as ChatGPT. We conducted a test by replacing ChatGPT with DeepSeek-V3 in MERIC, and found that MERIC can still identify all the input features as we reported in Table 1.

8 Related Work

Using machine learning methods for program optimization have been exploited for decades[17, 35, 39]. As large language models become mature and powerful, an increasing number of studies are exploring the potential of applying these models to program optimization problems. Ma *et al.* [21] conducted a comprehensive study to evaluate the capabilities of LLMs for code analysis, especially focusing on the ability to comprehend code syntax and semantic structures. Another work conducted by Tian *et al.* [32] presents an empirical study of ChatGPT's potential on the tasks of code generation, program repair, and code summariziation. They both conclude that while LLMs can comprehend basic code syntax, they are somewhat limited in performing more sophisticated analyses. The MERIC framework mitigates this limitation because of its compiler-guided reductive scheme which offers supplementary semantic structures and enriches the contextual understanding.

In software engineering, Feng and Chen [9] use LLMs to replay Android bug automatically. LLMs have been trained on source code including CodeBERT [10], CodeT5 [38], CodeGen[25] and CODELLAMA[29]. They are trained to perform multiple tasks including code search, code summarization, and documentation generation. Chris and other[3] use LLMs to output a list of compiler options to best optimize the program. Xia and others[40] use LLMs as an input generation and mutation engine to produce diverse and realistic inputs for universal fuzzing.

There have been several prior efforts on integrating machine-learning-based adaptation on OpenMP through a programming interface. Liao *et al.* [18] propose model-driven adaptive OpenMP extension automatically chooses the code variants. However, the selection of important variables is manually done. Kadosh *et al.* [15] explore the application of Transformer-based models to assist in OpenMP parallelization tasks. They do no automatic input characterizations.

9 Conclusion

This study is the first known exploration on leveraging LLMs to clear barriers for identifying critical input features for input-centric program optimization. It shows that combining LLMs with a novel reductive scheme (RACL) can effectively address the scalability challenges to LLM for code analysis and optimizations. By avoiding manual efforts and heavy profiling, the approach makes it possible for compilation workflow to seamlessly incorporate input characterizations, opening numerous opportunities for inputs-aware optimizations.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344, and was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) program, under SC-21. LLNL-CONF-868729.

This material is based upon work supported by the National Science Foundation (NSF) under Grant Nos. CNS-2312207 and OAC-2417850, the National Institutes of Health (NIH) under Grant No. 1R01HD108473-01, and the U.S. Department of Agriculture (USDA) under Grant No. P24-001771. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Energy, NSF, NIH, or USDA.

The United States Government retains, and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

References

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 158–165.

[2] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794. doi:10.1145/2939672.2939785

[3] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062* (2023). doi:10.48550/arXiv.2309.07062

[4] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[5] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[6] Daniel Sleator Davy Temperley and John Lafferty. 2016. Link Grammar. https://www.link.cs.cmu.edu/link/.

[7] RFV Der Wijngaart and H Jin. 2003. Nas parallel benchmarks, multi-zone versions. *NASA Advanced Supercomputing Division Ames Research Center, USA* (2003), 94035–1000.

[8] Dimitri van Heesch. 1997. Doxygen. https://www.doxygen.nl/.

[9] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3608137

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020). doi:10.48550/arXiv.2002.08155

[11] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023). doi:10.48550/arXiv.2312.10997

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[13] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 42–55. doi:10.1145/3625549.3658657

[14] Yunlian Jiang, Eddy Z Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. 2010. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 248–256. doi:10.1145/1772954.1772989

[15] Tal Kadosh, Nadav Schneider, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Advising openmp parallelization via a graph-based approach with transformers. In *International Workshop on OpenMP*. Springer,

3–17. doi:10.1007/978-3-031-40744-4_1

[16] LangChain, Inc. 2024. Map-Reduce: summarize long texts via parallelization. https://python.langchain.com/docs/tutorials/summarization/.

[17] Hugh Leather and Chris Cummins. 2020. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 1–8. doi:10.1109/FDL50818.2020.9232934

[18] Chunhua Liao, Anjia Wang, Giorgis Georgakoudis, Bronis R de Supinski, Yonghong Yan, David Beckingsale, and Todd Gamblin. 2021. Extending OpenMP for machine learning-driven adaptation. In *International Workshop on Accelerator Programming Using Directives*. Springer, 49–69. doi:10.1007/978-3-030-97759-7_3

[19] Andreas Löbel. 1997. *Optimal vehicle scheduling in public transit.* Ph. D. Dissertation.

[20] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* 125 (2021), 743–757. doi:10.1016/j.future.2021.07.021

[21] Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. 2023. The scope of chatgpt in software engineering: A thorough investigation. *arXiv preprint arXiv:2305.12138* (2023).

[22] Feng Mao, Eddy Z Zhang, and Xipeng Shen. 2009. Influence of program inputs on the selection of garbage collectors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 91–100. doi:10.1145/1508293.1508307

[23] Gleison Souza Diniz Mendonça, Chunhua Liao, and Fernando Magno Quintão Pereira. 2020. AutoParBench: a unified test framework for OpenMP-based parallelizers. In *Proceedings of the 34th ACM International Conference on Supercomputing*. 1–10. doi:10.1145/3392717.3392744

[24] NASA Advanced Supercomputing (NAS) Division. 2024. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html.

[25] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022). doi:10.48550/arXiv.2203.13474

[26] OpenWhisk. 2017. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[27] Apache OpenWhisk. 2017. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openWhisk-works.

[28] Yue-Hong Qian, Dominique d'Humières, and Pierre Lallemand. 1992. Lattice BGK models for Navier-Stokes equation. *Europhysics letters* 17, 6 (1992), 479.

[29] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023). doi:10.48550/arXiv.2308.12950

[30] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[31] Xipeng Shen and Feng Mao. 2008. Modeling relations between inputs and dynamic behavior for general programs. In *Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers 20*. Springer, 202–216.

[32] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938* (2023). doi:10.48550/arXiv.2304.11938

[33] Kai Tian, Yunlian Jiang, Eddy Z Zhang, and Xipeng Shen. 2010. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 125–139. doi:10.1145/1869459.1869471

[34] Kai Tian, Eddy Zhang, and Xipeng Shen. 2011. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 445–462. doi:10.1145/2048066.2048103

[35] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808* (2021). doi:10.48550/arXiv.2101.04808

[36] Truefoundry. 2023. Benchmaring Llama2-70B. https://www.truefoundry.com/blog/benchmarking-llama-2-70b/.

[37] VideoLAN. 2013. x264. https://www.videolan.org/developers/x264.html.

[38] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021). doi:10.48550/arXiv.2109.00859

[39] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. doi:10.1109/JPROC.2018.2817118

[40] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. *Proc. IEEE/ACM ICSE* (2024). doi:10.1145/3597503.3639121

[41] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. 2010. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 115–126. doi:10.1145/1810085.1810104

[42] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. 2019. Enabling runtime spmv format selection through an overhead conscious method. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 80–93. doi:10.1109/TPDS.2019.2932931

Received 2024-11-15; accepted 2025-03-06