A Study of Using LLM to Clear Barriers for Input-Centric Program Optimizations

Xiangwei Wang*, Xinning Hui*, Chunhua Liao+, Xipeng Shen*
* North Carolina State University
+ Lawrence Livermore National Lab

May 2024

Abstract

Input-centric program optimization is a pivotal approach to enhancing software performance by focusing on the relations between program inputs and program behaviors. Despite its promise, the field has struggled with significant challenges, primarily the difficulty of identifying critical input features from the complex inputs. Traditional methods, depending on manual analysis or statistical techniques, are labor-intensive and require many detailed profilings runs. This paper introduces a novel framework leveraging compiler-assisted Large Language Models (LLMs) to automate the identification of critical input features, aiming to clear these barriers and open new potential in program optimization.

The new solution, named MERIC (LLM-enabled reductive input characterising tool), uses a call-graph based reductive scheme (RAL) and a template-based prompt generation method (TPG) to overcome the scalability and other limitations of LLM in program code analysis. The application to eight programs demonstrates its efficacy in identifying critical input features. Predictive models built on those input features achieve an average accuracy of 93.1% in adaptive OpenMP parallelization tasks, helping produce near optimal shortest job schedules, and making the controllers in an open-source state-of-the-art serverless platform save 50-60% resource usage while delivering 20-30% performance improvement.

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 commandline 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 [Tian et al.(2010)]. The idea 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 [Tian et al.(2011)], the selection of the more suitable Garbage Collection [Mao et al.(2009)], GPU code optimizations [Zhang et al.(2010)], and so on.

Despite the promise of the concept 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. The existing approaches are either resorting to manual efforts [Shen and Mao(2008)] or statistical methods [Tian et al.(2010), Jiang et al.(2010)]. The latter requires extensive detailed profiling runs of a program on many different inputs. None of these approaches are desirable, making input-centric program optimizations difficult to adopt in practice.

In this work, we propose to address the long-lasting 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 part of the inputs is read and how it is 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, may provide the key to the input characterization problem.

At the core of our proposed compiler-guided LLM are two techniques: reductive analysis with LLM (RAL), and template-based prompt generation for LLM-based code analysis (TPG).

(1) RAL is designed to address the scalability limitation of LLMs. Here, the scalability

	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

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

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). Techniques (e.g., retrieval augmented generation [Gao et al.(2023)] and map-reduce [lan([n. d.])]) have been developed to go around the limit by retrieving the tokens most relevant to a request. But those techniques could leave out some important information, which is even more concerning for program analysis because those solutions were designed for natural languages and are oblivious to program structures (e.g., calling relations between functions).

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.

RAL solves the challenges by taking a reductive strategy. The strategy consists of reductions in two dimensions. (i) The first dimension is in program behaviors. The RAL-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 [Jiang et al.(2010)], 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. RAL 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 explosion. By reducing the focus of LLM to one function each time, RAL 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.

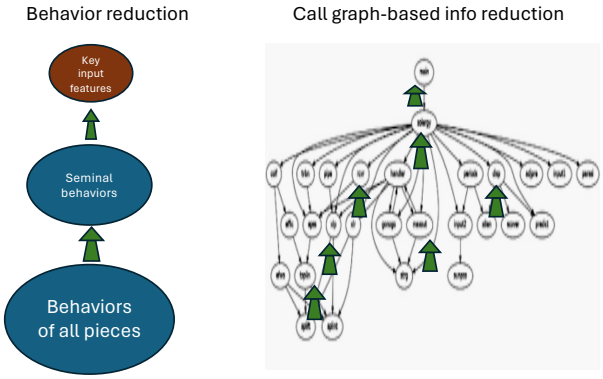

Figure 2: Reductive strategy employed by RAL

(2) TPG is designed to exert the potential of LLM for producing high-quality answers to requests in this problem domain. LLM is powerful but can give quality answers only if it is used well—using

the right prompts and providing needed guidance. Through many trials and errors, we create a set of templates and tools for generating prompts for this problem domain. The design of the templates follow a series of principles (detailed later) and observe the special features of code analysis and input characterizations. TPG is able to generate high-quality prompts automatically for each program, helping tap into the full potential of LLM.

With RAL and TPG, we create LLM-enabled reductive input characterising tool (MERIC), the first end-to-end 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 output the key feature values from an input to a program.

To test the efficacy of MERIC, we apply it to eight programs in various domains with 694–3740 lines of code in each. We test the usefulness of the characterized input features by MERIC in three uses, runtime optimizations of OpenMP parallelism, shortest-job-first (SJF) scheduling, and serverless scheduling. 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 parallelism or running time of the program of interest. The results show that the models can attain an average accuracy of 93.1% for OpenMP parallelism and 90.4% for predictive running times. The SJF schedules on its predicted times are on average within 2% of the length of the SJF schedules based on actual execution times. When applied to serverless computing, the predictions makes the controllers in an open-source state-of-the-art serverless platform save 50-60% resource usage while delivering 20-30% performance improvement.

Overall, this paper makes the following contributions:

- It presents the first compiler-guided LLMs for automatic input characterization of programs.
- It proposes a novel RAL scheme for addressing the scalability issues of LLMs for input characterization, which is potentially applicable to other code analysis problems as well.
- It shows TPG an effective way to generate prompts for LLMs to better analyze programs.
- It contributes the first end-to-end tool MERIC and empirically confirms the efficacy of the proposed techniques.

2 MERIC Overview and A Running Example

This section presents the overall architecture and workflow of MERIC for input characterization. It also presents a small program as a running example to facilitate the follow-up detailed explanation of the techniques.

2.1 Architecture and Workflow of MERIC

As shown in Fig.3, MERIC consists of three main components: *Preparation Processor*, *Code Analyzer* and *Extraction Module Creator*. Through a compiler-based tool, the *preparation processor* in MERIC 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*

[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.

traverses the reversed (modified) call graph by following the RAL scheme. It gets the seminal behaviors of the functions one by one by leveraging LLM on the prompts generated with TPG, passes the key insights from callees to callers through the way, 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.

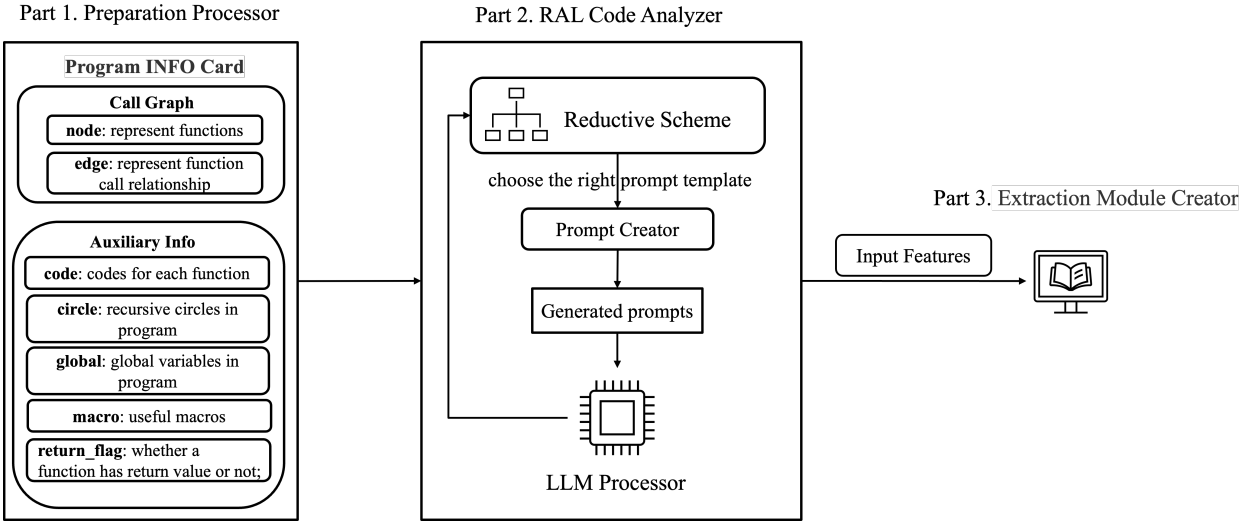

Figure 3: The architecture of MERIC framework.

2.2 A Running Example

To help the following explanation, we provide a running example as shown in Figure 4. 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*, which gets its value in function *main* from the largest number in the input file. 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 determines the running time of this program include the number of integers and the max value in the input file.

We next explain each component of MERIC in detail.

3 Preparation Processor

In the preparation phase, the preparation processor gathers the essential information from the source code and organizes them into program INFO card, a JSON file. It does it through an existing compiler-based tool, *doxygen*.

The program INFO card contains two parts: *call graph* and *auxiliary info*. Call graph includes information about nodes and edges within a directed graph, where each node symbolizes a function and each edge pointing from one node (caller) to another (callee) denotes a calling relation between

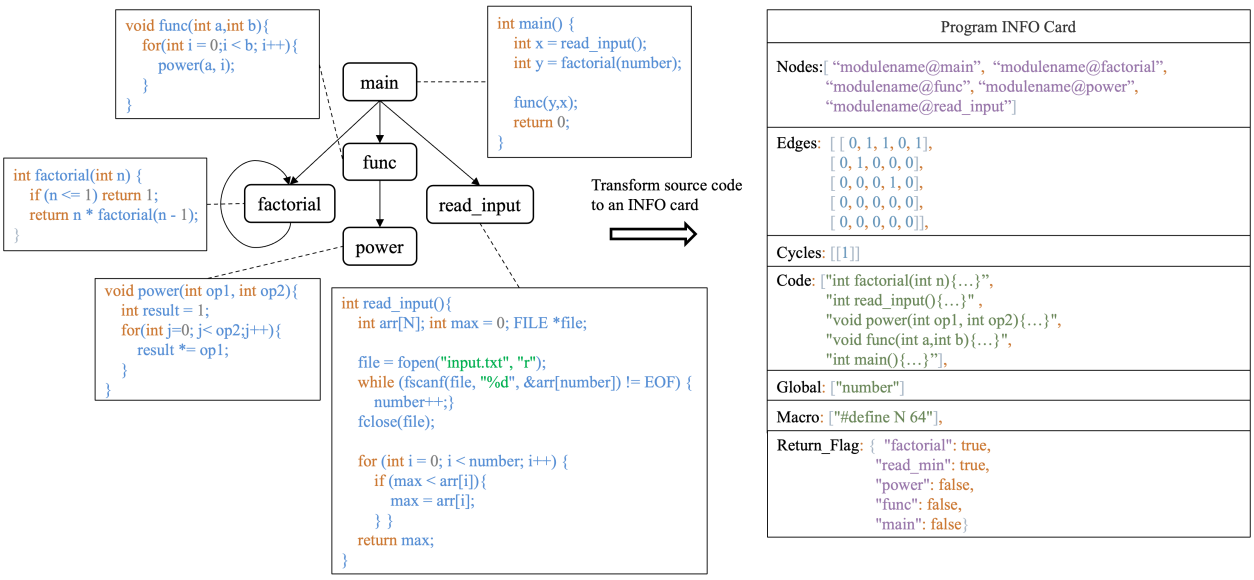

Figure 4: Left: a running example with its call graph. Right: the program INFO card of the program in JSON. Transform the source code of the example to program INFO card in preparation phase.

the two nodes. In the program INFO card, the “Nodes” field is a list containing the full-path names (to avoid namesakes) of all functions; the “Edges” field is a matrix with binary entries: entry (i,j) is 1 if function i calls function j. In Figure 4, entry (0,1) is 1, which indicates that index-0 node (”main”) calls index-1 node (”factorial”).

The other fields in program INFO card are auxiliary fields, containing the supplementary information of the program that are useful for constructing the prompts to LLMs. The fields are as follows:

- “code”: To facilitate the analysis process, we create a list to store each function’s source code. This setup allows the code analyzer to efficiently retrieve the relevant code snippets for each function. Using the same index, we can find the source code of the corresponding function in the node list. In our example, for instance, node[0] is ”main” function. Thus, code[0] has the source code for ”main” 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 Fig.4, this field is [[1]], incidating that there is only one recursion in the program and it is a self recursion on node[1] (factorial). We will explain the treatment to recursions in the next section.

- “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.

4 RAL Code Analyzer

The RAL code analyzer in MERIC takes the program INFO card as input and outputs the characterized input features. Its core consists of two techniques, RAL and TPG. This section first explains the overall workflow used by the analyzer, and then explains how RAL addresses the complexities from recursions, uses LLMs, represents key info of a function, and propagates the info across call graphs.

4.1 Overall Algorithm

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 to be used as the knowledge base to hold all the information the code analyzer needs 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 funInfo 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.

Note that we classify nodes in a call graph into three categories, leaf nodes, root node (usually the *main* function), and others which are called non-main caller nodes. As we will see later, *GetAnalyRes* works differently for the nodes in different categories. 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.

We illustrate how this iterative process works for our example program in Fig.5. Fig.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 Fig.5 (a). The analysis result includes seminar behaviors in the function and how program inputs influence that function and so on. We will explain it in detail in Section 4.3. 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 step b), the analyzer processes function "power" and added the results into the INFO set of its caller "func". It sees that "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"

Algorithm 1 RAL-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**

```
funInfo = initFunInfo(C);
G, TerminationCond = ElimateCycle(G);
for T in TerminationCond do
    node = T['function']
    Appendinfo(funInfo[node],T);
end for
Q = FindLeafNodes(G)
cnode = Q.headnode
AnalyRes = ∅
while N-Q do
    AnalyRes[cnode] = GetAnalyRes(cnode, funInfo[cnode]);
    for node in N-Q do
        if E(node, cnode) == 1 then
            Appendinfo(funInfo[node],AnalyRes[cnode]);
            node.count++;
            if node.count == node.outdegree then
                Q.append(node)
            end if
        end if
    end for
    move cnode to next node in Q;
end while
```

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 LLM include two behaviors. One is the value assigned to the global variable "number"; the reason LLM 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 seminal behaviors "number" and "x", LLM identifies two 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.

This reductive scheme is based on the function call relations and the definition of seminal behaviors. The seminal behaviors of a caller function are not only determined by its own source code but also by its callee functions. When LLMs analyzes the caller function's source code, without the info of the callee functions, the LLMs would not be able to analyze the caller well. The passing of the analysis results of the callees to caller in MERIC addresses the issue. One may wonder whether the passed analysis results would keep growing as the process moves up the call graph. It is not an issue because MERIC automatically controls the size and drops less important results along the way. It is discussed in the 'Size Control' part in Section 4.4. This process of reduction ensures that by the time the analysis reaches the main function, the identified seminal behaviors effectively encapsulate the important seminal behaviors of the entire program. In the actual implementation, while we focus on finding seminal behaviors and then input features most critical at the program level, we also preserve the analysis results of individual functions. This not only facilitates thorough examination but also supports other potential uses.

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 remainder of this section.

4.2 Treating Recursions

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. Consider a cycle as A -¿ B -¿ A. To put any of them into the processing queue (lines 17 in Algorithm 1), it would require all its callees to be processed. So A waits for B to be processed, while B waits for A to be processed, hence a cyclic dependence.

Our solution is to break the cycle but have the key info preserved with the help of LLM. Specifically, it sets 0 to the value of the edge that points to the function that has the lowest index in the cycle. Before removing any edge, it checks whether the cycle still exists based on the current edge matrix, as it may have already been broken while the algorithm works on other cycles. But before breaking the cycle, it first uses LLMs to figure out the termination condition of the recursion and then preserve the info in one of the involved function's INFO card. Specifically, it aggregates the source code of all the functions involved in the cycle together and integrates it into the prompt template as shown in Fig.6 and send the prompt to LLM. Notice that the prompt also asks which variables control the termination for the recursion. 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 cycle 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 may get only some partial results in B and complains that "To decide what seminal behaviors are in function A, I need the source code of this function". Those results 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 and A's code as well as termination condition together to do the behavior analysis.

4.3 Results Representation

As mentioned, for scalability, RAL limits the scope of view of LLM to the code of only 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 find out and 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 recorded analysis results is the record on the return values of a function. The returned values of a function could be used by its callers and hence influence the caller's behaviors. So it is important to record the return values and how these values are influenced by program inputs. That is what this field is about. The LLM is asked to find the info out when analyzing a function as shown by Figure 7 (c), and the response is kept as part of the analysis result of that function.

4.4 Results Propagation with Conversion and Size Control

The propagation of the analysis results from callees to a caller is simply by RAL 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. RAL asks LLMs to do that by including that request in the prompt template to a caller function (Figure 8). 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 (the `position` field mentioned in Section 4.3).

We use two functions in our running example to illustrate the mapping process. As shown in Fig.9, function "main" is the caller of the function "func". In the callee function, variable "b" is recognized as func's seminal behavior. Since the callee's seminal behavior should be considered when analyzing the caller, the code analyzer passes func's seminal behaviors with their chosen reasons to the main function. When LLM reads function main's source code and its callee's seminal behaviors, it finds that variable "b" in the callee function controls the number of loop iterations and is crucial to the execution time. Thus, the code analyzer should keep "b" in the main function's semimal behaviors. However, if the LLM doesn't have extra info, when it analyses the "main" function, it sees only the "main" function code, and would not be able to decide which variable in "main" corresponds to variable "b".

RAL addresses the issue by keeping the exact position information for callee's seminal behaviors. Since variable "b" is a parameter in function "func", as shown in Fig.9 (b), RAL marks its position as the second parameter. With that info passed to the caller's INFO card, when the code analyzer meets the caller function, LLM is able to replace the name of callee's seminal behavior with the correct caller's variable. In this case, variable "b" is correctly mapped to variable "x".

Size Control Theoretically speaking, the seminal behaviors of a function shall be reported by the LLMs as seminal behaviors of the caller as well because the execution of the callee is part of the execution of the caller. The analysis result of a function could grow larger and larger as RAL 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 (e.g., 8192 for ChatGPTv4-8k) 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 automatically selective in reporting seminal behaviors. The auxiliary info that the analysis result records, such as the reasons for selecting a behavior as a seminal behavior (Section 4.3), offers LLMs the basis for selecting the more important behaviors. We have observed two common cases where LLMs drop a callee's seminal behavior when analyzing the caller. One case is that in viewing the code of caller function, the callee's seminal behavior's impact to the program behavior of interest (e.g., run time) become less important. Another case is that the value of the callees' seminal behavior is determined by a caller's.

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. It again uses LLMs to do that. Figure 10 (a) shows the prompt template. 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 RAL is because the RAL 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 order-related complexity and how the info tracking in RAL helps, we draw on a simple example in Fig.11. Based on algorithm.1, the code analyzer starts with functions "r1" and "r2". The analysis results of "r1" and "r2", including *input order* and *input influence*, are propagated to "main". Based on the prompts of the "main" function in Fig 10, LLMs merge the results from "r1" and "r2", and summarize the input order and influence for the entire program as the result in Figure 11 shows. Note that it is able to include the conditions that affect the reading order in the summary, which helps the construction of the correct feature extraction module later.

Another point worth mentioning is that 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. 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.

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 features of the input described in text, "the number of integers in the file" and "the maximum value in the file". In case where the reading order of inputs depend on certain conditions on some value read earlier from the input, the mapping result by LLMs will include those conditions; see Figure 10 for an example. In the input order of the "main" function, LLMs summarize conditions to call the function "r1" or "r2".

Generating Feature Extraction Module The final job of RAL code analyser is to generate a input feature extraction module. The produced module is a code snippet. When applied to a new program input, it can extract values of input features. It can be used to run before the target program starts. Whenever a new input comes in, it can quickly extract the necessary input features and apply a prediction model to predict the program's behaviors and hence guide runtime optimizations. (We discuss the construction of prediction models in the next section.)

Because the previous step already produces the descriptions of the key input features, it seems that 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. Figure 10 (b) shows the basic LLM template we designed for this phase.

But some input features identified by RAL analyser are massive objects in an input on which LLM has a hard time to extract some concise features. A common case is that the values of the massive objects determine the convergence of an algorithm. 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. We give the module generator an option to create an inspection tool with LLMs. As the bottom bullet in the prompt template in Figure 12 shows, this option allows the LLM to produce a variant of the original program, which samples several iterations of the program while outputting the values of some variables (residuals) that could shed light on the progress (e.g., convergence rate) of the execution. The right side in Figure 12 is an example inspection code produced by LLM: LLM opens the source code, revises the iterative part of the program with a fixed number of iterations and inserts monitoring code into the loop.

5 Design of Prompt Templates

High-quality prompts are important for LLMs to give high-quality results. As we have mentioned, The strategy taken by MERIC is to use a set of prompt templates to ensure the quality of the prompts given to LLMs. The template generator in RAL uses the info in the INFO cards to instantiate the templates into actual prompts.

We have already shown the main templates in the previous section (Figures 7, 8, 10). In this part, we provide some insights we obtained in our explorations in designing those templates (some of the insights drew inspirations from prior experience [Bsharat et al.(2023)]):

- To ensure clarity and prevent any misunderstanding, it's essential to introduce the context and purpose of the content before incorporating it into prompts. For instance, before embedding code snippets or specific content into a prompt, a clear explanation should be given to outline what the content represents. Conversely, inserting content without prior explanation or offering clarification only after that can lead to confusion, reducing the effectiveness of the LLM's responses and potentially leading to misinterpretations of the task requirements. In our prompt templates in Fig 7, 8 and 10, we provide clear definitions for the used concepts, such as seminal behaviors, input influence and return value.

- Break down complex tasks into a sequence of simpler prompts in an interactive conversation. Or in a single prompt, use leading words like "Solve the problem in the following steps" and

give the clear instructions about what to do in each step. It is also the basic idea of Chain-of-thought[Wei et al.(2022)]. For example, to recognize seminal behaviors for a non-main caller function, we define two steps in the prompt in Fig 8. In Step1, we require LLMs to do mapping and in step2 we ask LLMs to recognize seminal behaviors.

- Implement example-driven prompting (Use few-shot prompting[Brown et al.(2020)]). By providing LLMs with a small set of carefully selected examples or scenarios that illustrate the task at hand, we can effectively teach the model how to respond to similar prompts. In the prompt template in Fig 12, to create the inspection tool, we provide an example on how to generate inspectors for a program with convergence loops.

- Format answers and limit token size. In our MERIC framework, most of outputs from LLMs serve as the immediate results. To ensure these results are both efficient and cost-effective, it's important to standardize the format of our answers, preferably in a JSON format, and impose a limit on the length of these answers' length, like "return your answers in 100 tokens". Moreover, we've observed that the response time from LLMs is influenced by the length of its answers. Shorter answers are typically generated and delivered more quickly. By setting a token size limit, we can expedite the receipt of analysis results, enhancing the overall pace of our workflow.

In addition, to enhance the efficacy of prompt templates, we find that it is good to devise some supplementary tasks to support the primary objective. For the tasks in Fig 7, for instance, we instruct LLMs to summarize the reasons behind selecting a variable for seminal behaviors. Additionally, we establish guidelines to record "pos" for various types of variables (local, global, parameter). Incorporating these instructions into prompts improves the efficacy of LLMs in identifying seminal behaviors.

6 Predictive Models and Correctness Concerns

This section briefly discusses how one may use the identified input features in input-centric optimization and the correctness concerns.

Predictive Model Construction To use the identified input features in input-centric optimization, one would need to build a predictive model to predict the behaviors of the program of interest 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 7), we use XGBoost [Chen and Guestrin(2016)], a Random Forest library [Liaw et al.(2002)], as the primary approach for it is lightweight and capable. Other machine learning methods could also be used.

Correctness Concerns A risk of using LLMs is that it may not be always reliable, for its statistical nature, hallucination and other issues. The compiler-guidance employed by MERIC gives it some help. But overall, it is hard to make LLM's output always error-free. It is however less of a concern for input-centric optimization, because in this context, LLMs don't directly change the original program, but offer only ways to extract key input features. The optimizations are still done by compilers or runtime. They leverage the results from LLMs as hints; if the hints are wrong, the optimization may not be very effective, but the damage can be well controlled by the design of the possible options of the optimizations as we demonstrate in the evaluation (Section 7).

7 Evaluation

We implement MERIC based on Doxygen and the out-of-box ChatGPT v4 APIs without LLM fine-tuning. To evaluate the efficacy of MERIC, we test three input-centric optimizations based on the inputs features characterized by MERIC on eight programs.

7.1 Tested Optimizations

Use Case 1: OpenMP Parallelism As a popular parallel programming models, OpenMP [Dagum and Menon(1998)] automatically parallelizes a code region based on the directives (pragmas) added to the code region. 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 popular job scheduling algorithms used in operating systems and other runtime 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—it is what our experiment is about in this use case.

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 [ope([n. d.]), Ope([n. d.])]. 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 [Hui et al.(2024)]. 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 input changes, 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.

7.2 Methodology

Benchmarks Because our optimizations include OpenMP optimizations, we use NAS Parallel Benchmarks(NPB) [Löff et al.(2021)] as our main programs as it is a popular OpenMP benchmark. To increase the coverage, we added two extra programs MCF and LBM from SPEC CPU® 2017[Standard Performance Evaluation Corporation(2017)]. MCF has a larger code size, and LBM

Table 1: Benchmarks and Input Features by MERIC

Program	Program Description	LOC	Program Input	#Inputs	Input File Size	Input Features by MERIC
CG	Conjugate Gradient, irregular memory access and communication	694	First line: three integers to define sparse matrix size (NA), nonzero numbers, shift; NA lines: each line has one value representing row pointer; The rest of the file: each line has two numbers: - the first number represents the column index for this element - the second number represents the value for this element	200	157KB∼990MB	NA,NONZERO,SHIFT values from inspector (rnorm1,rnorm2,rnorm3)
MG	Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive	1032	First line: an integer defining *Level*; Second line: three values to define three dimensions of the grid; Third line: an integer defining # iterations; The rest of the file: each line stores the values of a grid point	56	12KB∼1.1G	Level, Three Dimensions, Iterations
FT	Discrete 3D fast Fourier Transform	967	First line: three integers to define three dimensions for problem size; Second line: one integer to define number of iterations; The rest of the file: two arrays with size NX*NY*NZ, each value in an array is a complex number in a (real, imag) pair	256	48KB∼2.6G	Three Dimensions, Iterations
BT	Block Tri-diagonal solver	2963	First line: two numbers to define # iterations and timestamp(dt); Second line: three integers to define three dimensions; The rest of the file: each line has 5 values representing the attributes of a grid point	138	72KB ∼1.6G	Iterations, Dt, Three Dimensions
SP	Scalar Penta-diagonal solver	2781	same as BT	144	72KB ∼1.6G	Iterations, Dt, Three Dimensions
LU	Lower-Upper Gauss-Seidel solver	3475	same as BT	140	72KB ∼1.6G	Iterations, Dt, Three Dimensions
MCF	Single-depot vehicle scheduling in public mass transportation	3740	First line: the numbers of timetabled trips and dead-head trips; the following lines are - for each timetabled trip its starting and ending time; - for each dead-head trip its starting and ending timetabled trip and its cost;	52	126KB ∼788MB	number of timetabled trips number of dead-head trips
LBM	Lattice Boltzmann Method to simulate incompressible fluids in 3D	1350	Several command line arguments: lbm <grid size><time steps><0: ldc, 1: channel flow> Description of the arguments: <grid size>: three integers for SIZE_X, SIZE_Y and SIZE_Z; <time steps>: number of time steps that should be performed before the results are stored; <0: ldc, 1: channel flow>: two basic simulation setups lid-driven cavity or channel flow	180	–	grid dimensions time steps simulation setup

has more complexities in its inputs. Table 1 lists all the eight programs[2], source code size, inputs and their sizes, and the input features identified by MERIC.

We collect 52–256 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. Specifically, we collect those inputs by either searching the real uses of the corresponding applications or deriving the inputs after gaining enough understanding of the benchmark through reading its source code and example inputs. 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. These are hyper parameters for CG program. And when the original CG program executes, it randomly generate a simple sparse matrix. We collect real-world sparse matrices in SuiteSparse Matrix Collection [Davis and Hu(2011)] and transform them into the appropriate format which CG program can directly use.

Machines We use a local Linux machine to collect execution time. And the hardware used in serverless computing experiment is a 64-node cluster. Table 2 reports the details of the local machine and the machine in the cluster.

Comparison Counterparts We compare MERIC with the *map-reduce* method, an approach taken by langchain [lan([n. d.])] to deal with large documents. This method decomposes a large document into smaller segments for analysis. In the "map" phase, each segment is independently

[2]Three of the programs (BT, SP, and LU) each contain a convergence loop. To make them more realistic, we add convergence checks into the loops' termination conditions.

Table 2: Hardware Configuration

	Cluster Node	Local Machine
CPU Model	AMD EPYC 7302P 16-Core Processor	Intel(R) Xeon(R) CPU E5-2630 v4 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

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. Figure 13 shows the prompts used in the map and reduce phases.

Predictive Models and Measurement As mentioned in Section 6, we use XGBoost [Chen and Guestrin(2016)] as the predictive model for behavior predictions. For a given program, we collect or generate many different program inputs and use our extraction module to obtain the values of input features. To build the predictive models, we collect data by executing the program on its inputs (and with the various parallel settings in the OpenMP optimization case) and measuring its running times. We use cross validation [Hastie et al.(2009)] (80% data for training and 20% for testing) for evaluation. By default, each reported result is the average of 20 repeated measurements.

7.3 Identified Input Features

The rightmost column in Table 1 shows the input features identified by MERIC. We underline features which are also identified by the map-reduce method.

For all the programs, the features by the map-reduce method all include "every element in the input file" as a feature, along with some non-input features, such as "the number of iterations in the 'adi' function". Directly using those features did not lead to any usable predictive models. We hence added a filtering step to remove those useless features. The remaining features are subsects of those identified by MERIC, marked with underlines in the rightmost column of Table 1.

We next report the benefits brought by the input-centric optimizations enabled by the identified input features.

7.4 OpenMP Parallelism Optimization

We show the prediction accuracy of the best parallel settings in Table 3. The predictor built on the features from MERIC shows high prediction accuracies, from the lowest accuracy 89.3% for MG to the highest accuracy 97.5% for LU. In comparison, the predictors on the features from the map-reduce method achieves significantly lower accuracies on most of the programs. The comparison indicates the higher quality of the input features identified by MERIC.

Figure 14 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.

Table 3: 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%	90.4%	95%
Map-Reduce	71.3%	71.4%	93.4%	75.7%	65.6%	90.9%	90.4%	55.6%

Table 4: Prediction Error for Execution Time

Program	CG	MG	FT	BT	SP	LU	MCF	LBM
MAPE	0.063	0.105	0.062	0.109	0.079	0.109	0.129	0.111

The results confirm that clear advantages of the results from MERIC over map-reduce. For interest of space, we will focus on MERIC results in the following sections.

7.5 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. We show the prediction mean absolute percentage errors (MAPE) for each program in Table.4. For most programs, the error rates hover around 0.1, ranging from the lowest at 0.063 for CG to the highest at 0.129 for MCF.

Figure 15 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 results show that the predicted times by MERIC can let the SJF schedule produce near optimal wait time, which can be many times shorter than the input-oblivious case.

7.6 Serverless Computing

In this experiment, the optimized controller [Hui et al.(2024)] in OpenWisk schedules jobs by leveraging the estimated job running times. We create a serverless function for each of the eight programs. The workloads in the experiments are a series of 800 calls to those functions with each call taking a randomly chosen input. We examined the traces published by Azure [Shahrad et al.(2020)] 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 16 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 16 (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".

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.

7.7 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. 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. But as Table 1 shows, for all the programs except *CG*, 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. *CG* is special because several of its input features are extracted through an inspector which could take several iterations of the main loop in the program. It is however possible to hide the overhead by, for instance, using the first several iterations of the main loop for inspection rather than using a separate inspection program. Because overhead treatment is an orthogonal problem to input characterization, we leave it out of this work.

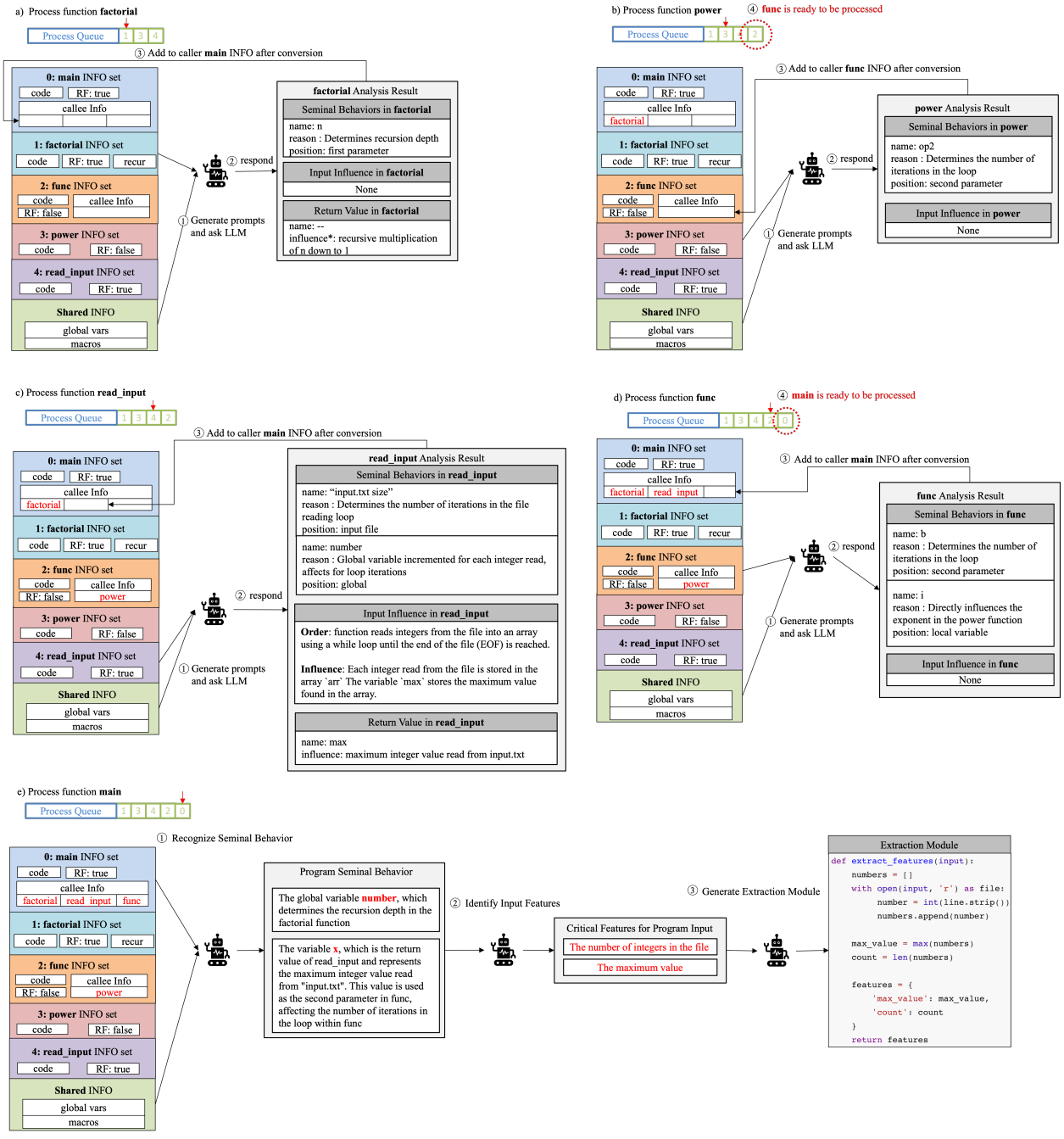

Figure 5: Illustration of how RAL code analyzer works on the running example in Figure 4 step by step. The left hand of sub-figure a)-d) shows each function's INFO set. They are integrated into prompts. The right hand of sub-figure a)-d) shows analysis result from LLMs. In e), main function is a special 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. (*influence* in the return value box is about how program input decides the value of the return variable; "input.txt size" in (c) is a built-in name for the size of the input text.)

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

Figure 6: The used prompt and LLM's answer to deal with the recursion in the example program. LLM identifies the termination condition involving "n".

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".

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

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 code:

========== start of the code ==========
{code}
========== end of the code ==========

These are global variables in this program: {global}.
These describe how the return variables of callee functions get value: {return_info}
These show how program inputs influence variables in callee functions: {input influence}

For each function called within this function, their identified seminal behaviors are {callee seminal behaviors}

Based on information above, following these steps:
step1: If a seminal behavior is a function parameter in the callee function, replace the "name" in the seminal behavior with the corresponding variable name used in the caller function.
step2: Find seminal behaviors in this function's code that most significantly affect the execution time.

Format the identified seminal behaviors into a JSON object with the following structure and guidelines…

b) Prompt to analyze input influence

Read the following code:

========== start of the code ==========
{code}
========== end of the code ==========

These describe how the return variables of callee functions get their values: {return_info}
These are the influences of program inputs on variables in callee functions： {input influence}

You should follow these steps:
Step 1. Summarize the order in which the program reads inputs and how program inputs (such as which lines) influence variables in this function,
Step 2. Combine the order of input and the influences from callee functions with those of this function, ensuring the correct execution order is maintained.

If this function and its callee functions doesn't read program input, just return "None".

c) Prompt to track return value

Read the following code:

========== start of the code ==========
{code}
========== end of the code ==========

These describe how the return variables of callee functions get value: {return_info}
These are the influences of program inputs on variables in callee functions： {input influence}

This function has a return variable. Find which part of input decide the value of the return variable.
Format the answer with a JSON format:
…..

Figure 8: Prompt templates used for non-main caller functions.

(a)

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

    func(y,x);
    return 0;
}
```

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

(b) With parameter position record, map 'b' to correct caller's variable 'x'

Seminal Behaviors in **func**	Seminal Behaviors in **main**
name: b reason : Determines the number of iterations in the loop position: second parameter	name: x reason: Controls the number of loop iterations
name: i reason : Directly influences the exponent in the power function position: local variable	.....

Figure 9: An example to illustrate the reason to map callee functions' parameters to caller functions' variables. (a) A call relationship between function main() and function func(); (b) With the mapping info, the analyzer correctly maps callee's seminal behavior 'b' to variable 'x' and gives the correct reason.

a) Prompt to identify input feature

Read the following code:

========== start of the code ==========
{code}
========== end of the code ==========

These are the program's recognized seminal behaviors and the reasons for choosing them:
{recognized seminal behaviors}.

These describe how program input influence variables in callee functions: {input influence}.

Step1. Summarize the order in which the program reads inputs
Step2. Summarize how seminal behaviors are influenced by program input.

b) Prompt to create extraction module

These are program seminal behaviors and how program inputs influence recognized seminal behaviors :
{input influence}.
We define the features of program input which influence seminal behaviors as input features.

Write a small program that can extract the values of input features for a given program input.

Figure 10: Extra content in the prompt templates for the "main" function. a) shows prompt template to identify input features. b) shows prompt template to generate extraction module. Contents in braces should be filled with the LLM's answer from the previous task.

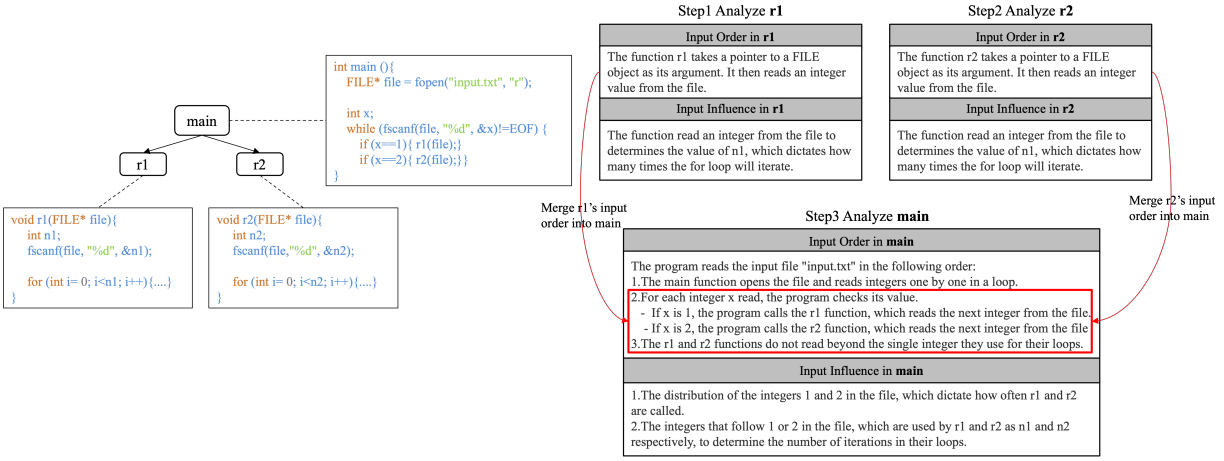

Figure 11: An example to illustrate the merge process for program inputs. The left directed graph is a simple call graph with four functions. The right graph shows the code analyzer merges input order and input influence based on function call. At the end of the process, after analyzing main function, the code analyzer should understand how the program read contents from inputs.

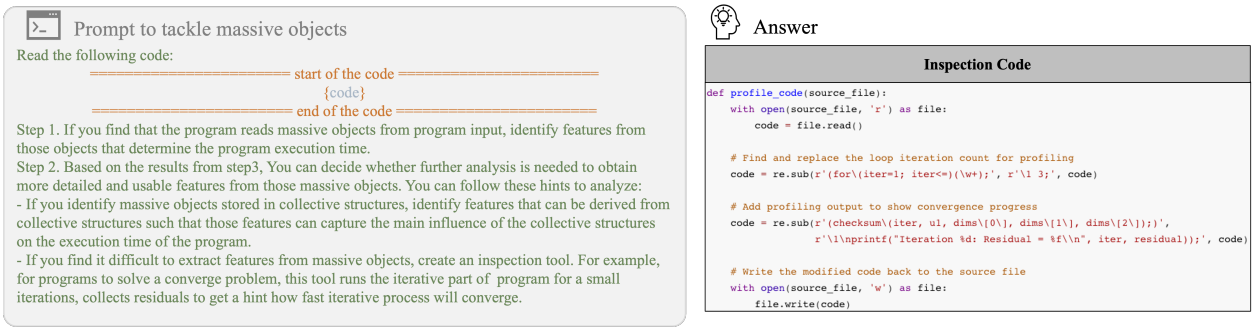

Figure 12: Left: Prompt template for creating the inspection tool. Right: An example inspection tool produced by LLMs.

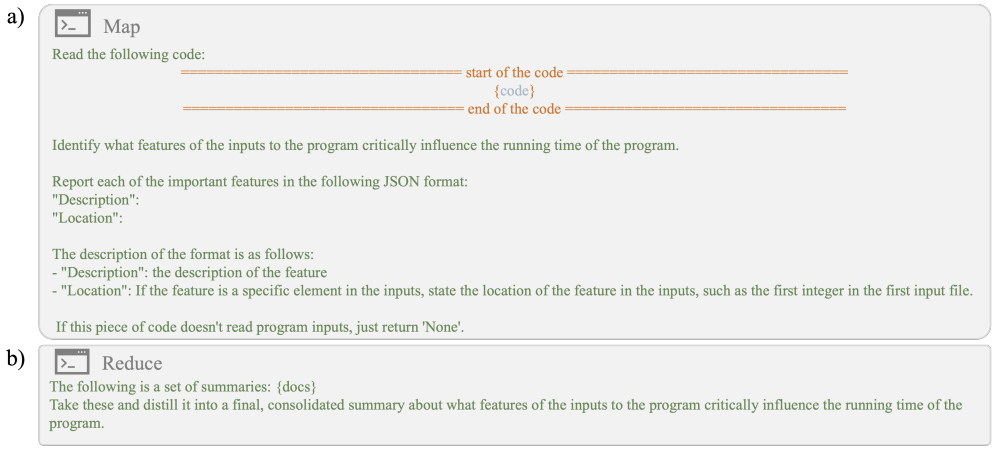

Figure 13: The prompts used in the map-reduce scheme in langchain.

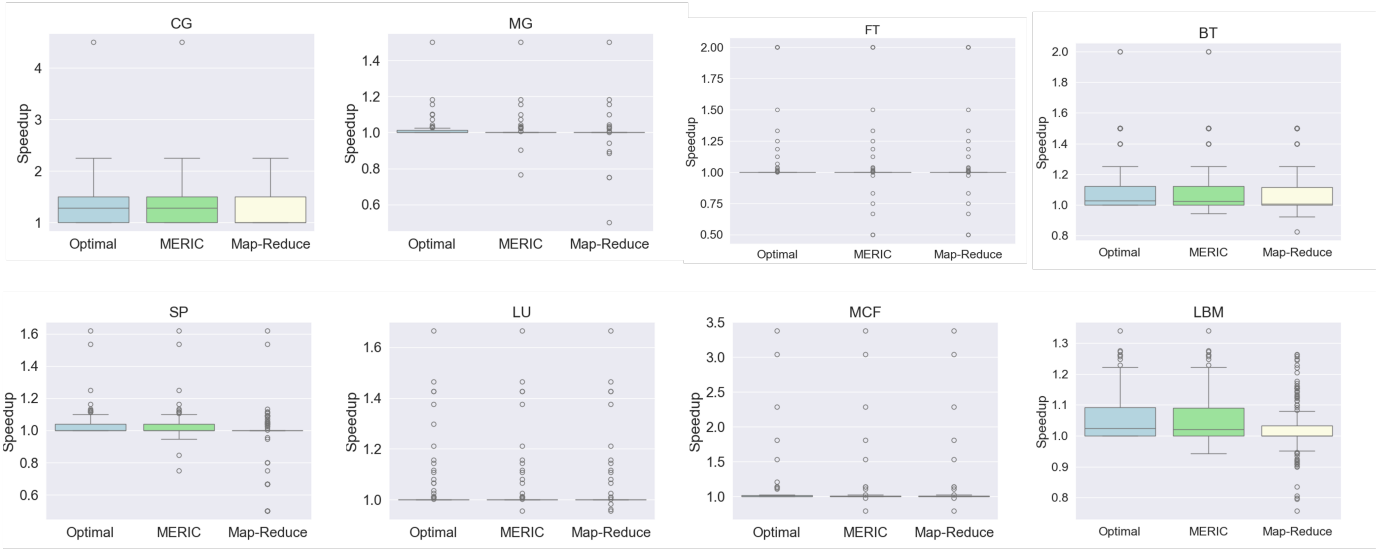

Figure 14: Speedups brought by OpenMP parallelism optimizations.

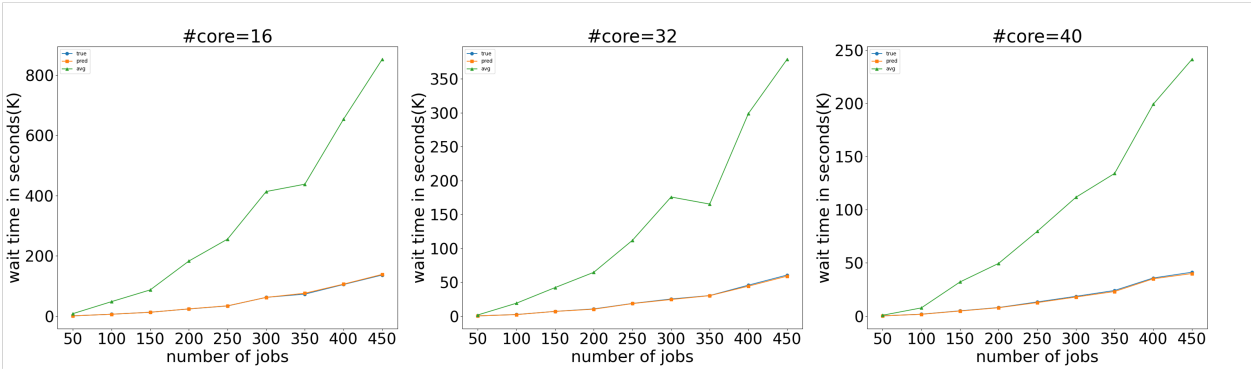

Figure 15: Total wait time in SJF schedules. The 'true' curves and 'pred' curves are largely overlapped.

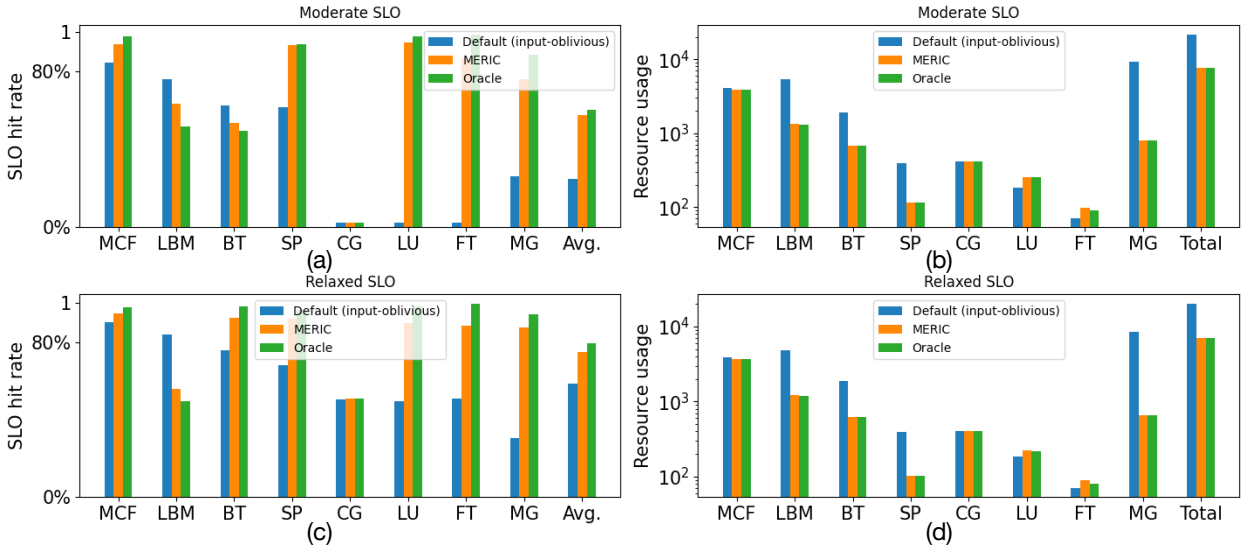

Figure 16: The performance and resource usage in serverless computing when the workload is steady. (a) and (c) show the SLO hit rate in moderate and relaxed SLO level, respectively. The higher, the better. (b) and (d) show the resource usage. The lower, the better.

8 Related Work

Using machine learning methods for program optimization have been exploited for decades[Wang and O'Boyle(2018), Leather and Cummins(2020), Trofin et al.(2021)]. 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.* [Ma et al.(2023)] 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 which include abstract syntax trees (AST), control flow graphs (CFG), and call graphs (CG). Another work conducted by Tian *et al.* [Tian et al.(2023)] 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 for several reasons: 1) It employs a reductive scheme based on the call graph, which offers supplementary semantic structures, enriching the contextual understanding. 2) The framework includes carefully crafted prompt template designs that guide LLM towards generating more focused and relevant outputs.

Many works focus on using LLM to solve the specific problems in program optimization and software engineering. In our work, we explore the possibility to use LLM to identify input features. Feng and Chen [Feng and Chen(2024)] use LLM to replay Android bug automatically. Like our work, they leverage few-shot learning and chain-of-thought reasoning to design prompts, and facilitate LLMs without any training and hard-coding effort. Meanwhile, many language models have been trained on source code including CodeBERT [Feng et al.(2020)], CodeT5 [Wang et al.(2021)], CodeGen[Nijkamp et al.(2022)] and CODELLAMA[Roziere et al.(2023)]. They are trained to perform multiple tasks including code search, code summarization, and documentation generation. LLMs trained on source code have also been used for more specific tasks. Chris and other[Cummins et al.(2023)] use LLM to output a list of compiler options to best optimize the program. Xia and others[Xia et al.(2024)] use LLM as an input generation and mutation engine to produce diverse and realistic inputs for universal fuzzing. Xia and others[Xia et al.(2023)] also evaluate LLMs for automated program repair. These efforts are continuously expanding the application of LLMs to software engineering.

There have been several prior efforts on integrating machine-learning-based adaptation on OpenMP through a programming interface. Liao *et al.* [Liao et al.(2021)] propose model-driven adaptive OpenMP extension automatically chooses the code variants. However, the selection of important variables is manually done. Kadosh *et al.* [Kadosh et al.(2023)] explore the application of Transformer-based models to assist in OpenMP parallelization tasks. None of them does automatic program input characterizations. These studies however do not account for input sensitivity.

9 Conclusion

This study is the first known exploration on leveraging LLM to clear barriers of identifying critical input features for input-centric program optimization. Through the development and implementation of the MERIC framework, we demonstrate the potential of LLMs to systematically recognize input features from analyzing program source code. Our approach, which combines LLMs with a novel reductive scheme (RAL), addresses the scalability challenges to LLM for code analysis and optimizations. Through a carefully designed template-based prompt generation scheme (TPG), it effectively capitalizes on the power of LLM for code analysis.

The evaluation conducted across a suite of benchmark programs underscores the efficacy of

MERIC in identifying key input features. By facilitating the construction of predictive models, our framework has shown to effectively guide program optimizations such as adaptive OpenMP parallelization and runtime scheduling. The predictive models achieve the average accuracy of 93.1% for adaptive OpenMP parallelization tasks, near-optimal SJF schedules, and 20-30% performance enhancement and 50-60% resource savings in serverless computing.

This work, for the first time, demonstrates the feasibility of a fully automatic way to characterize the important inputs features for an unknown program without detailed profilings, making automatic input-centric optimization a promising direction. Its methods and insights in combining the strengths of compilers and those of LLMs and overcoming their limitations can potentially benefit other code analysis and optimizations.

Acknowledgement

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-2312207, and Department of Energy (DOE) under Grant No. DE-SC0021293. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

References

[Ope([n. d.])] [n. d.]. Apache OpenWhisk. How OpenWhisk works. https://github.com/apache/openwhisk/blob/master/docs/about.mdhow-openWhisk-works.

[lan([n. d.])] [n. d.]. LangChain. Summarization. https://python.langchain.com/docs/use$_c$*ases/summarization*/.

[ope([n. d.])] [n. d.]. OpenWhisk. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[Brown et al.(2020)] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[Bsharat et al.(2023)] Sondos Mahmoud Bsharat, Aidar Myrzakhan, and Zhiqiang Shen. 2023. Principled Instructions Are All You Need for Questioning LLaMA-1/2, GPT-3.5/4. *arXiv preprint arXiv:2312.16171* (2023).

[Chen and Guestrin(2016)] 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.

[Cummins et al.(2023)] 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).

[Dagum and Menon(1998)] 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.

[Davis and Hu(2011)] 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.

[Feng and Chen(2024)] 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.

[Feng et al.(2020)] 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).

[Gao et al.(2023)] 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).

[Hastie et al.(2009)] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

[Hui et al.(2024)] Xinning Hui, Yuanchao Xu, Zhishan Guo, and Xipeng Shen. 2024. ESG: Pipeline-Conscious Efficient Scheduling of DNN Workflows on Serverless Platforms with Shareable GPUs. arXiv:2305.02522 [cs.DC]

[Jiang et al.(2010)] 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.

[Kadosh et al.(2023)] 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.

[Leather and Cummins(2020)] 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.

[Liao et al.(2021)] 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.

[Liaw et al.(2002)] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[Löff et al.(2021)] 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.

[Ma et al.(2023)] 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).

[Mao et al.(2009)] 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.

[Nijkamp et al.(2022)] 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).

[Roziere et al.(2023)] 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).

[Shahrad et al.(2020)] 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.

[Shen and Mao(2008)] 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.

[Standard Performance Evaluation Corporation(2017)] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017 Benchmark Suite. https://www.spec.org/cpu2017/.

[Tian et al.(2023)] 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).

[Tian et al.(2010)] 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.

[Tian et al.(2011)] 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.

[Trofin et al.(2021)] 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).

[Wang et al.(2021)] 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).

[Wang and O'Boyle(2018)] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[Wei et al.(2022)] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[Xia et al.(2024)] 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).

[Xia et al.(2023)] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[Zhang et al.(2010)] 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.