CBANA: A Lightweight, Efficient, and Flexible Cache Behavior Analysis Framework

Qilin Hu, Yan Ding, *Member, IEEE*, Chubo Liu, *Member, IEEE*, Keqin Li, *Fellow, IEEE*, Kenli Li, *Senior Member, IEEE*, and Albert Y. Zomaya, *Fellow, IEEE*

***Abstract*—Cache miss analysis has become one of the most important things to improve the execution performance of a program. Generally, the approaches for analyzing cache misses can be categorized into dynamic analysis and static analysis. The former collects sampling statistics during program execution but is limited to specialized hardware support and incurs expensive execution overhead. The latter avoids the limitations but faces two challenges: inaccurate execution path prediction and inefficient analysis resulted by the explosion of the program state graph. To overcome these challenges, we propose CBANA, an LLVM- and process address space-based lightweight, efficient, and flexible cache behavior analysis framework. CBANA significantly improves the prediction accuracy of the execution path with awareness of inputs. To improve analysis efficiency and utilize the program preprocessing, CBANA refactors loop structures to reduce search space and dynamically splices intermediate results to reduce unnecessary or redundant computations. CBANA also supports configurable hardware parameter settings, and decouples the module of cache replacement policy from other modules. Thus, its flexibility is established. We evaluate CBANA by using the popular open benchmark PolyBench, graph workloads, and our synthetic workloads with good and poor data locality. Compared with the popular dynamic cache analysis tools Perf and Valgrind, the cache miss gap is less than 3.79% and 2.74% respectively with over ten thousand data accesses for the synthetic workloads, and the time reduction is up to 92.38% and 97.51% for the multiple-path workloads. Compared with the popular static cache analysis tool Heptane, CBANA achieves a time reduction of 97.71% while ensuring accuracy at the same time.**

***Index Terms*—Cache behavior modeling, path analysis, path selection, static analysis.**

Manuscript received 23 September 2023; revised 9 May 2024; accepted 27 May 2024. Date of publication 19 June 2024; date of current version 9 August 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB0300300; in part by the National Natural Science Foundation of China under Grant 62072165; Grant 62202154, and Grant 62225205; in part by the Science and Technology Innovation Program of Hunan Province under Grant 2021RC3063 and Grant 2023GK2003; in part by the Science and Technology Program of Changsha under Grant kh2301011; and in part by China Scholarship Council (CSC). Recommended for acceptance by A. Karanth. *(Corresponding authors: Yan Ding; Chubo Liu.)*

Qilin Hu, Yan Ding, Chubo Liu, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, and also with the National Supercomputing Center in Changsha, Hunan 410082, China (e-mail: hql@hnu.edu.cn; ding@hnu.edu.cn; liuchubo@hnu.edu.cn; lkl@hnu.edu.cn).

Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, also with the National Supercomputing Center in Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Albert Y. Zomaya is with the School of Computer Science, University of Sydney, Sydney, NSW 2006, Australia (e-mail: albert.zomaya@sydney.edu.au).

Digital Object Identifier 10.1109/TC.2024.3416747

I. Introduction

ADVANCEMENTS in chip design have widened the performance gap between memory and processor [1], [2]. The recent Intel Core i9 processor has around 4 ns of calculation latency, while memory latency exceeds 90 ns [3]. To bridge the gap between calculation latency and memory access latency, cache has been introduced as an important component [4], [5], [6], [7]. Cache miss serves as a critical criterion for analyzing the execution performance of a program. A lower number of cache misses usually indicates the overhead reduction in data movement and thus significantly improves execution performance.

Depending on whether the program is executing during the analysis, existing cache miss analysis can be categorized into dynamic analysis and static analysis as illustrated in Fig. 1. Dynamic analysis can usually be further classified into run-time profilers and simulators. Run-time profilers [8], [9], [10] estimate cache misses by sampling data in Performance Monitoring Counter (PMC). Simulators [11], [12], [13], [14], [15] realize fine-grained memory access trace through dynamic binary instrumentation (DBI) during the program execution on simulated hardware/system. Static analysis methods [16], [17], [18], [19], [20] generally abstract static information from programs and construct a model of execution flows to analyze the cache misses. Although current dynamic and static analyses provide various ways to evaluate cache misses, they still have individual limitations and inefficiencies.

Dynamic analysis is highly limited to specialized hardware and also incurs expensive execution overhead. For example, a run-time profiler samples a portion of cache behavior and then estimates the entire program performance, which is limited to the number of PMCs and the sampling frequency. Simulators avoid hardware constraints caused by insufficient number of PMCs through fine-grained simulation of instruction streams, but the modification and interception of instructions incur additional run-time overhead. In addition, the dynamic analysis scheme requires multiple executions of the program to cover

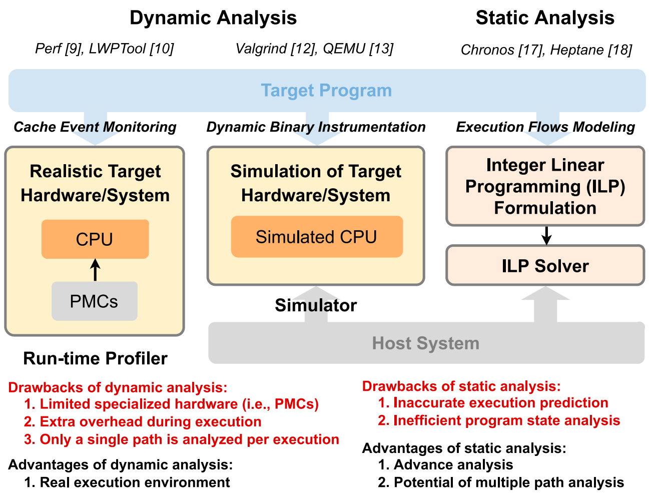

Dynamic Analysis
Static Analysis
Perf [9], LWPTool [10]
Valgrind [12], QEMU [13]
Chronos [17], Heptane [18]
Target Program
Cache Event Monitoring
Dynamic Binary Instrumentation
Execution Flows Modeling
Realistic Target Hardware/System
CPU
PMCs
Simulation of Target Hardware/System
Simulated CPU
Integer Linear Programming (ILP) Formulation
ILP Solver
Simulator
Host System
Run-time Profiler
Drawbacks of dynamic analysis:
1. Limited specialized hardware (i.e., PMCs)
2. Extra overhead during execution
3. Only a single path is analyzed per execution
Advantages of dynamic analysis:
1. Real execution environment
Drawbacks of static analysis:
1. Inaccurate execution prediction
2. Inefficient program state analysis
Advantages of static analysis:
1. Advance analysis
2. Potential of multiple path analysis

Fig. 1. Design trade-off between different cache miss analysis methods.

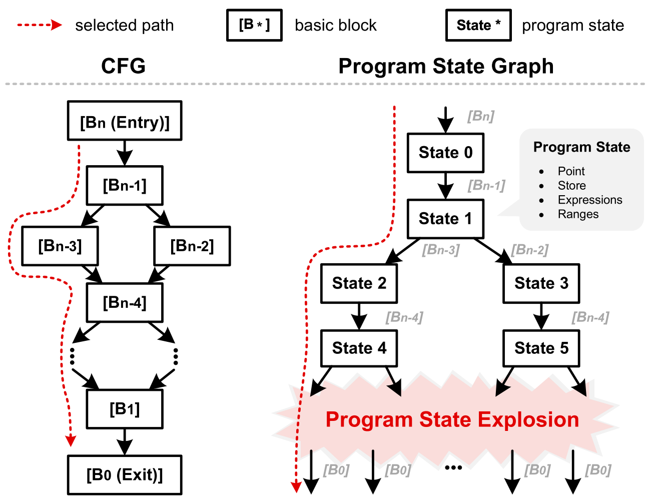


Fig. 2. Program state graph is generated by the path sensitive analysis of CFG. Further splitting of program states may cause the explosion.

multiple or even entire execution paths in the program, which further leads to analysis lag.

In contrast, static analysis provides the potential to analyze cache misses without the hardware constraints and execution overhead. However, most existing static analysis schemes only analyze cache misses on a single path or the worst-case execution path due to the challenges of inaccurate execution prediction and inefficient program state analysis. Many path selection schemes use branch prediction to decide which branch should be taken. However, the accuracy of branch prediction that uses heuristic methods is probabilistic [21]. Furthermore, large-scale programs face the risk of state explosion when generating the program state graph. As shown in Fig. 2, a program can be represented as a Control Flow Graph (CFG). The utilization of language constructs such as loops may cause an exponential increase in the number of program states [22], thus incurring the state explosion and leading to an abrupt interruption of analysis.

In this work, we try to take advantage of static analysis while tackling its challenges. We design CBANA, a static cache behavior analysis framework, to analyze the cache misses of the program executed on systems with a cache architecture. Accurate execution prediction is achieved by leveraging input-aware path analysis. Loop refactoring and dynamic splicing of intermediate results ensure efficient data stream generation and lightweight cache miss calculation. Flexible cache behavior modeling is provided by configurable hardware parameter settings and decoupling the module of the cache replacement policy from other modules. Overall, the main contributions of CBANA are as follows:

- We present CBANA, a static cache behavior analysis framework that overcomes the challenges of inaccurate execution path prediction and inefficient program state analysis.
- We enable an input-aware path analysis mechanism. It abstracts a program into multiple program points and introduces program states to describe the various conditions of a program point. Analyzing the program states ensures the accuracy of path selection.
- We propose a dynamic splicing scheme of intermediate results to reduce unnecessary or redundant analysis. The concrete data stream is generated after getting the execution path. For overlapping parts on different paths, the information inside a basic block can be reused.
- For the synthetic workloads with more than ten thousand data accesses, our evaluation shows that compared with the popular dynamic cache analysis tools (i.e., Perf and Valgrind), the cache miss gap is less than 3.79% and 2.74%, respectively. In addition, CBANA achieves up to 92.38% and 97.51% time reduction when the multiple-path workloads are analyzed by 1000 sets of inputs. Compared with the popular static cache analysis tool Heptane, CBANA achieves a time reduction of 97.71% while ensuring accuracy at the same time.

This paper is organized as follows. Section II provides an overview of related work and highlights the characteristics of CBANA. Section III describes the organization and design details of our framework. Section IV covers the implementation of CBANA and presents the evaluation results. Finally, Section V concludes this paper.

II. Related Work

As mentioned earlier, existing methods for analyzing cache misses are generally categorized into dynamic analysis [9], [10], [12], [13], [23] and static analysis [17], [18], [19], [24] according to whether the source program needs to be executed during the analysis. As is shown in Table I, we compare CBANA with representative methods from various aspects, including the characteristics of the analysis methods, the ability of path analysis, the analysis granularity, and the cache architecture supported by these methods.

Dynamic analysis includes run-time profilers and simulators. Run-time profilers leverage a set of PMCs in modern processors to capture program run-time performance information [9], [10],

TABLE I
Comparison of the Techniques in Cache Miss Analysis

Category	Methods	Program Execution	Path Analysis	Limitations	Analysis Granularity	Supported Architectures (i.e., cache mapping, type, and replacement policy)
Dynamic	Perf [9]	Yes	N/A	1. Limited specialize hardware 2. Extra overhead during execution 3. Single path analysis per execution	Instruction	-
	LWPTool [10]				Line	
	RDX [23]				Program	
	Valgrind [12]				Function/Line	I/D/LL cache, LRU
	QEMU [13]				Instruction	SA, I/D cache, LRU/FIFO/Rand
Static	SPS [19]	No		1. Inaccurate execution prediction 2. Inefficient program state analysis	Loop	LRU
	Chronos [17]		Worst-case		Program	Direct/SA, I-cache, LRU
	Heptane [18]					SA, I/D cache, LRU/PLRU/FIFO/MRU/Rand
	Exact-CS [24]				Program/Fragment	FA/SA, I/D cache, LRU
	CBANA (this paper)		Input-aware	-	Program/Function/Line	FA/SA, D-cache, LRU/FIFO/MRU

Note: **FA** = fully associative, **SA** = set-associative, **I** = instruction, **D** = data, **LL** = last level, **LRU** = least recently used, **PLRU** = pseudo-LRU, **FIFO** = first in first out, **MRU** = most recently used, **Rand** = random, **N/A** = not applicable.

[25], [26]. PMCs collect cache-related events (e.g., L1-dcache-load-misses and L1-icache-load-misses) during the execution of a program. However, inadequate sampling frequency and the sharing of a single counter between multiple events can cause considerable errors [27], [28]. Jitter and noise during sampling can also introduce uncertainty in cache miss measurements [29]. Simulators achieve accurate cache behavior analysis by modeling hardware architecture [15]. For example, Cachegrind detects cache data interactions by simulating a machine with L1 and L2 cache. It is available in a framework for dynamic binary analysis called Valgrind [12]. A cache simulation module is also established in QEMU [13] to provide cache miss rate measurement. To alleviate the simulating overhead, CANAL [14] models the cache behavior by inserting a sequence of optimization (opt) passes before and after each Load/Store instruction during the program execution. Although the system modeling techniques are relatively well established, the running of the simulator may require significant computing resources. For instance, DBI involved in the simulation will modify the source code and introduce significant additional execution overhead.

Static analysis based on abstract interpretation uses mathematical semantics to describe the computing process of a program [16]. By maintaining an abstract domain at each fixpoint, the abstract properties of the program can be analyzed by the abstraction of the actual behavior. Heptane [18] is a representative approach that provides data address and cache analysis based on abstract interpretation. It is capable of determining the intervals of addresses by analyzing the content of each register with dataflow equation representation [30] and associating every memory reference with Cache Hit/Miss Classification (CHMC) under different cache architecture. Abstract interpretation enhanced with speculative execution has been implemented in many popular analysis tools, such as LLVM [31] to detect the number of cache misses on the path with Worst-case Execution Time (WCET) analysis [20]. Chronos [17] performs detailed micro-architectural modeling to generate WCET estimates of C programs. Both Heptane and Chronos use Integer Linear Programming (ILP) formulation to represent the execution time of the entire program and use ILP solvers to find WCET estimates. Cache persistence analysis is also studied as an important part of WCET analysis. Considering all memory accesses in a program or a fragment of a program, Exact-CS [24] introduces the first exact persistence analysis for caches with LRU replacement.

The LRU stack distance, also known as reuse distance, is a widely used definition that can be applied in both dynamic analysis and static analysis. It proves to be an effective criterion for recognizing code with poor data locality in LRU replacement algorithm [23], [32], [33], [34], [35], [36]. Stack distance measures the distinct data between two memory accesses and was first proposed by Mattson et al. [37]. To describe how close a set of data is accessed in a program, reference affinity is defined in [38]. It uses the stack distance as a reference to reorganize the array and the structure. To improve analysis efficiency, the approximate analysis of stack distance is proposed, which trades accuracy for efficiency [32]. An alternative method for measuring working-set locality is reuse time [39], which counts the number of accesses between two consecutive accesses to the same cache block. Static parallel sampling is proposed to predict miss ratio curves for complex loops and branches based on reuse time, which measures almost exact results as trace-based analysis [19]. A lightweight profiling tool RDX [23] integrates PMU event-based sampling and debug registers to analyze reuse distance and characterize program's locality during the execution. However, the reuse distance is designed for the LRU algorithm and cannot be applied to other cache replacement algorithms.

In view of the drawbacks of the existing approaches, CBANA is designed to pinpoint program performance bottlenecks in

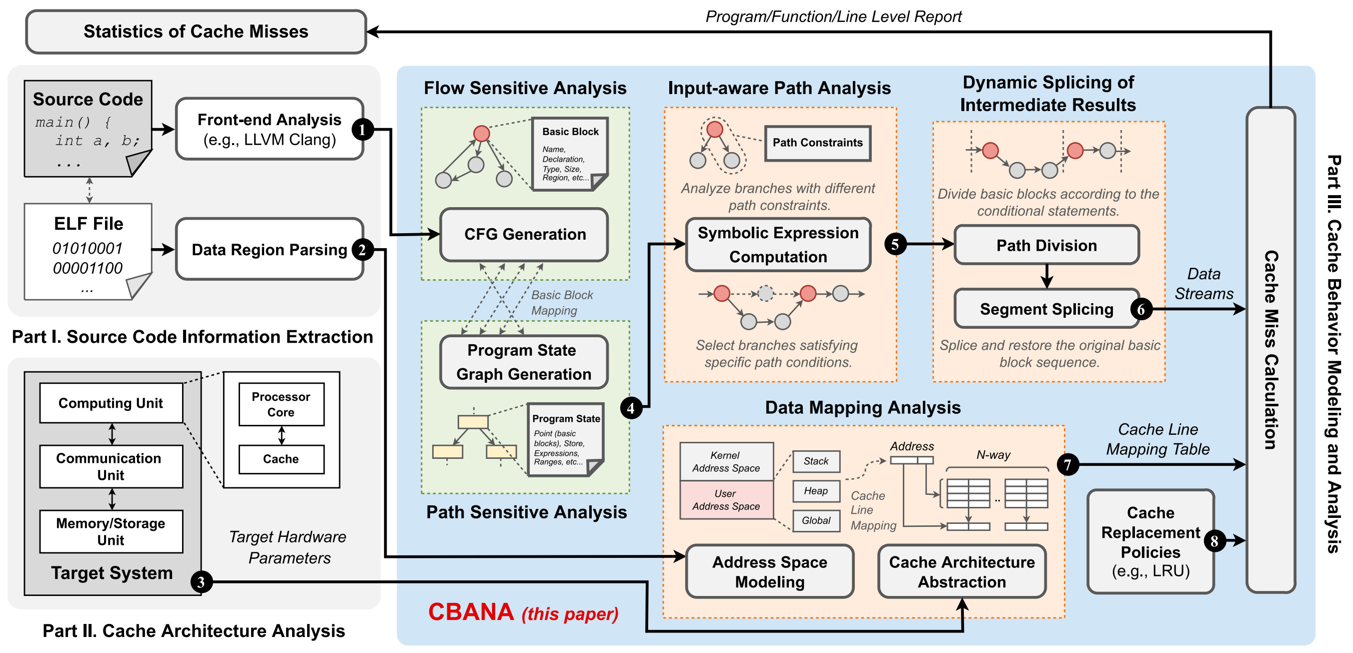

Statistics of Cache Misses
Program/Function/Line Level Report
Source Code
main() {
int a, b;
...
Front-end Analysis (e.g., LLVM Clang)
ELF File
01010001
00001100
...
Data Region Parsing
Part I. Source Code Information Extraction
Flow Sensitive Analysis
Basic Block
Name, Declaration, Type, Size, Region, etc...
CFG Generation
Basic Block Mapping
Program State Graph Generation
Program State
Point (basic blocks), Store, Expressions, Ranges, etc...
Path Sensitive Analysis
Input-aware Path Analysis
Path Constraints
Analyze branches with different path constraints.
Symbolic Expression Computation
Select branches satisfying specific path conditions.
Dynamic Splicing of Intermediate Results
Divide basic blocks according to the conditional statements.
Path Division
Segment Splicing
Splice and restore the original basic block sequence.
Data Streams
Cache Miss Calculation
Part III. Cache Behavior Modeling and Analysis
Computing Unit
Communication Unit
Memory/Storage Unit
Processor Core
Cache
Target Hardware Parameters
Target System
Part II. Cache Architecture Analysis
Data Mapping Analysis
Kernel Address Space
User Address Space
Stack
Heap
Global
Address
N-way
Cache Line Mapping
Cache Line Mapping Table
Address Space Modeling
Cache Architecture Abstraction
Cache Replacement Policies (e.g., LRU)
CBANA (this paper)

Fig. 3. Overview of CBANA. It mainly contains six modules: flow sensitive analysis, path sensitive analysis, input-aware path analysis, dynamic splicing of intermediate results, data mapping analysis, and cache miss calculation.

different paths of a program at a fine-grained granularity. As a static analysis scheme, CBANA is able to correlate cache misses to the location of variable access line-by-line. Besides, the input-aware path analysis and configurable cache architecture ensure the efficiency and flexibility of analyzing multiple paths in parallel. Meanwhile, dynamic splicing of intermediate results reduces the risk of program state explosion during static analysis and therefore ensures a lightweight cache miss calculation.

III. CBANA for Analyzing Cache Behavior

In this section, we present CBANA. First, we introduce the overall design of CBANA. Second, we show the design details of key components in CBANA.

A. Overall Framework Design

Fig. 3 illustrates the overview of the cache behavior analysis framework. The framework is mainly divided into three parts: Part I, source code information extraction, Part II, cache architecture analysis, and Part III, cache behavior modeling and analysis (CBANA in this paper). Part I and Part II mainly provide the program and hardware information required by Part III. In Part I, the source code is analyzed by a front-end analysis tool such as LLVM Clang and compiled into an ELF file to extract the source code information. In Part II, the cache architecture parameters of the target system are taken as inputs. In Part III, cache behavior is modeled according to the extracted information, and cache misses will be calculated according to the selected execution paths. Part III is mainly divided into six modules:

Flow Sensitive Analysis: It divides the statements into several basic blocks and generates CFG according to the control flow between basic blocks. The program point is defined as a node in CFG.

Path Sensitive Analysis: It correlates each program point with more than one program point and generates a program state graph to indicate realistic execution paths with path constraints. The program state will record the current related basic blocks and other detailed information.

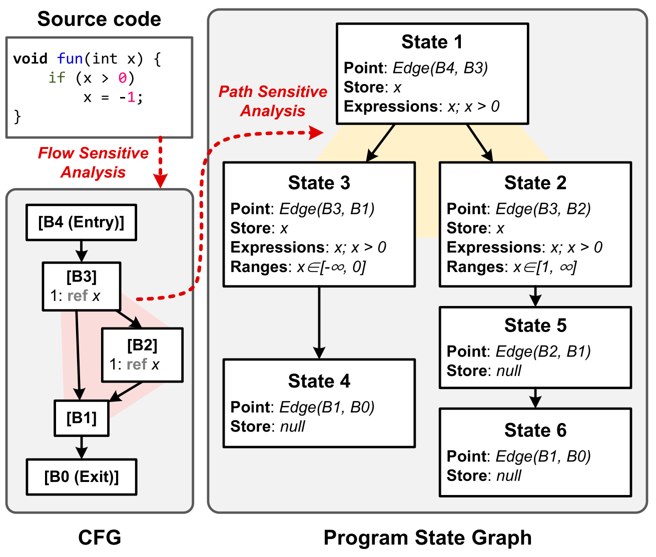
Input-aware Path Analysis: With the path information provided by the CFG and the program state graph, it is responsible for analyzing path constraints and selecting branches according to the computation results of symbolic expressions. Then it will generate a simplified basic block sequence.

Dynamic Splicing of Intermediate Results: It divides the simplified basic block sequence into multiple path segments according to the conditional statements. To relieve the overhead brought by analyzing large-scale loops, segment splicing restores the original basic block sequence with the intermediate results and the number of loop iterations. The data stream is generated from the sequence and internal information of basic blocks.

Data Mapping Analysis: It models the virtual address space by using the compilation information and cache architecture information to generate the cache line mapping table.

Cache Miss Calculation: It takes data streams, cache line mapping table, and cache related architecture parameters as inputs to estimate the number of cache misses for selected execution paths.

Overall, as shown in Fig. 3, the framework consists of eight stages (Stages ❶-❽). In Stage ❶, front-end analysis generates the CFG and program state graph. In Stage ❷, virtual address



```
void fun(int x) {
    if (x > 0)
        x = -1;
}
```

Fig. 4. Source code is divided into a number of basic blocks in CFG, and the different conditions of the expression in the program point further introduce distinct program states.

space is modeled to map the data into different cache lines in the subsequent steps by combining process address space in the operating system (OS) and parsing data regions. At the same time, in Stage ❸, the parameters of the cache architecture in the target system are taken as inputs for data mapping analysis. In Stage ❹, the input-aware path analysis determines the execution path by computing the symbolic expressions in the path constraints of each branch. In Stage ❺, each data stream is dynamically spliced according to the block sequence of the path. In Stage ❻, the data streams are taken as one of the inputs to cache miss calculation. In Stage ❼ and Stage ❽, the cache line mapping table and cache replacement policies are also taken as the inputs for cache miss calculation. After that, the cache misses of the corresponding path can be estimated.

Flow sensitive analysis and path sensitive analysis are used for the preprocessing of the whole program, which analyzes all of the basic blocks and corresponding program states. Input-aware path analysis provides accurate path selection and efficient program state analysis. CBANA also supports configurable cache architecture parameters, where the address space modeling promises the spatial locality involved in cache line replacement. Moreover, CBANA employs a decoupled design to support more potential types of high-level languages and cache replacement policies.

B. CFG and Program State Graph Generation

Flow Sensitive Analysis: The goal of flow sensitive analysis is to infer program control flow and generate CFG. We divide the statements into several basic blocks where the control flow can only enter from the first statement and come out from the last statement in a basic block. After the program is divided into basic blocks, CFG is built to indicate the control relationship between the basic blocks. Take the source code in Fig. 4 as an example, the program is divided into five basic blocks, and it starts at an entry $B4$ and ends with an exit $B0$. The conditional statement (i.e., *if*) in basic block $B3$ introduces two edges pointing to different basic blocks $B1$ and $B2$, which will introduce different program points.

Path Sensitive Analysis: To further correlate concrete inputs or classes of inputs with different execution paths, path sensitive analysis is done to generate the program state graph. The program state includes the accumulated execution situations when reaching the program point. Each program state owns four main attributes, which are *Point*, *Store*, *Expressions*, and *Ranges*. *Point* records the basic block associated with the current program state, *Store* records the effects of assignments in statements, *Expressions* records the detailed expressions, and *Ranges* is optional and records the states that satisfy the current branch. Fig. 4 shows the conversion of CFG into a program state graph. By evaluating the conditional statement, two program states are created with different path conditions *true* and *false*, which correspond to the two branches $x \geq 1$ and $x \leq 0$. Therefore, the program state graph may end with multiple exits.

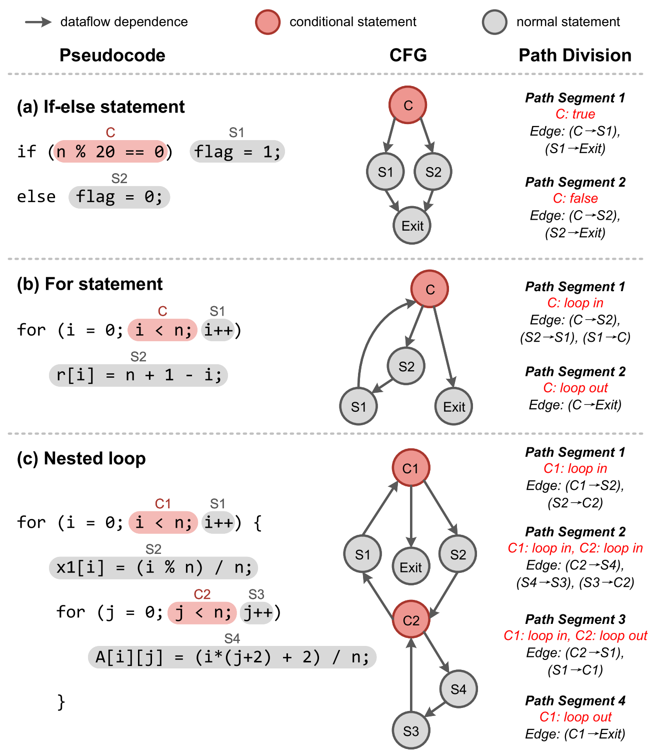
With the flow sensitive analysis and path sensitive analysis, we get the path constraints on the corresponding branches. Analyzing data and instruction streams is well-studied [11], [12], [17], [18]. However, these studies focus on analyzing a single path in the compiled file or the worst-case execution path. In this study, path sensitive analysis relates a path to a specific path constraint, where basic blocks on different paths can be analyzed simultaneously and independently. The expenditure of symbolic expressions is also well utilized. When the path constraints satisfy the current condition, path sensitive analysis further updates the path constraints according to the symbolic expression on the subsequent path. Therefore, the paths can be filtered according to the class of inputs to reduce unnecessary analysis. For example, in Fig. 5(a), if the variable n satisfies the statement $n\%20 == 0$, the condition corresponds to $(C \rightarrow S1)$ and $(S1 \rightarrow Exit)$. If the condition is false, it corresponds to $(C \rightarrow S2)$ and $(S2 \rightarrow Exit)$.

Based on the path constraints, input-aware path analysis will utilize the information from *Ranges* and *Point* to determine the path constraints that meet the conditions and decide the final execution path that matches the current input.

C. Input-Aware Path Analysis

Input-aware path analysis builds on the granularity of basic blocks and is used to determine which branch should be chosen for a particular input condition.

To select the exact path, variables with concrete inputs or classes of inputs in the current period are plugged into symbolic expressions of path constraints to decide which path to enter by analyzing the results of expressions. As shown in the Fig. 5(a), if the variable n is a multiple of 20, $(C \rightarrow S1)$ and $(S1 \rightarrow Exit)$ will be selected. For program states with multiple branches satisfying different conditions, analyzing the path constraints in parallel improves the efficiency of path selection. At this point, a complete sequence of basic blocks constrained by the input is



```
if (n % 20 == 0) flag = 1;
else flag = 0;
```

```
for (i = 0; i < n; i++)
    r[i] = n + 1 - i;
```

```
for (i = 0; i < n; i++) {
    x1[i] = (i % n) / n;
    for (j = 0; j < n; j++)
        A[i][j] = (i*(j+2) + 2) / n;
}
```

Fig. 5. Examples of input-aware path analysis from original code to CFG. The statements in pseudocode are represented as nodes in CFG, where the conditional statements incur different branches with distinct path conditions.

generated. However, the states of the variables within the basic block may affect the symbolic expressions in the conditional statements, traditional methods will repeat the program state within the loop at each round of loop iterations, leading to a program state explosion. To solve the problem of program state explosion, dynamic splicing of intermediate results is further introduced in CBANA.

D. Dynamic Splicing of Intermediate Results

As the symbolic expression of each variable is updated, the increase of program states will lead to the program state explosion. Specifically, the current program state will be forked into two different states when facing a conditional branch [40]. Therefore, the number of feasible paths will grow exponentially along with the expansion of the branch scale. CBANA solves the problem by recognizing and reusing the intermediate results in the source code. Take the definition of compile-time enumerable in SPS [19] as a reference, the data streams in the loops are compile-time enumerable when the expressions of the loop bounds, strides, branch predicates, and array subscripts contain only the loop index variables and constants. Based on the assumption of enumerable in the program, CBANA provides dynamic splicing of intermediate results of basic blocks.

Path Division and Segment Splicing: We define the basic blocks divided by the conditional statements as path segments, where a path segment is the basic unit of the dynamic splicing and consists of at least one basic block. To reduce the redundant program state and analyze the loops more efficiently, the basic blocks included in loops are fully expanded into the original sequence of basic blocks after the execution path is selected. Take Fig. 5(b) as an example, we analyze one iteration of the loop. Once the value of variable n is captured, the path segment 1 (i.e., $(C \rightarrow S2)$, $(S2 \rightarrow S1)$, and $(S1 \rightarrow C)$) will be spliced dynamically according to the number of original iterations. For nested loops, we first expand the inner loop and then the outer loop. As Fig. 5(c) shows, the basic blocks in the nested loop are divided into four path segments. Path segment 2 in the inner loop is firstly spliced according to the value of variable n, and then the segments (i.e., path segment 1, the spliced path segment 2, and path segment 3) included in the outer loop are further spliced totally according to the number of iterations.

Dynamic splicing of intermediate results avoids excessive concentration on the internal information of basic blocks during loop analysis or among different paths. When input-aware path analysis of multiple paths involves overlapped basic blocks, analysis at the granularity of basic blocks helps to reduce redundant and unnecessary information analysis. For the final data stream generation of multiple paths, we can reuse the internal information of previous basic blocks without reanalyzing the entire program.

E. Data Mapping Analysis

To guarantee the precision of cache miss analysis, we need to determine how the variables are mapped in the cache line. Using the cache line as the granularity of the cache behavior analysis enables better integration of cache analysis with the spatial locality of the data. As shown in Fig. 6, the source code is compiled into an ELF file and further parsed into virtual address space distribution in a real runtime environment. Furthermore, the virtual address space is divided into global, stack, and heap regions. Then the data will be mapped into different cache lines according to the cache parameters. The mapping information will be recorded in the mapping table for the cache behavior analysis. For different variables, if the mapping table records the same *Tag* and *Index*, it indicates that they are in the same cache line. Therefore, we can analyze the mapping situation of variables in fully associative and set-associative based on the obtained virtual addresses.

Understanding the distribution of virtual address space benefits the procedure of cache line mapping. The operating system provides a separate virtual address space for each process in order to manage memory more efficiently and securely [41]. The mapping principles of these regions are usually as follows:

- `Code (.text)`: Contains executable instructions.
- `Initialized data (.data)`: Saves initialized global and static variables, which can be further divided into the read-only region and initialized read-write region.
- `Uninitialized data (.bss)`: Stores global and static variables that are not explicitly initialized in the source code.
- `Stack`: Located in the high part of user address space and grows towards the lower address as the function is called.
- `Heap`: Grows from low addresses to higher addresses and is maintained by a dynamic memory allocator in the form

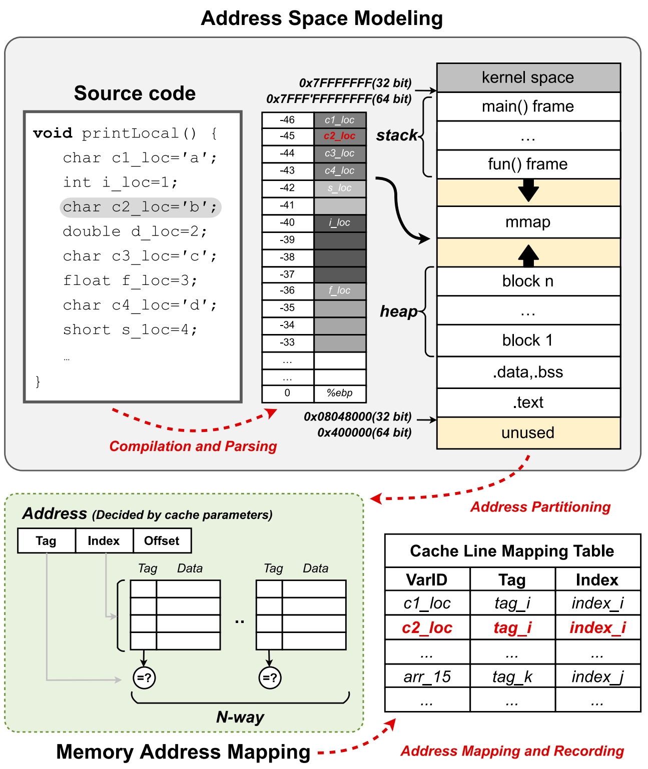


Fig. 6. Memory address mapping. Firstly, the address space is modeled by parsing the ELF file compiled from the source code. Secondly, addresses are partitioned based on the cache architecture. Thirdly, *Tag* and *Index* are mapped and recorded into the cache line mapping table according to the addresses.

of an implicit free list. The memory in heap region can be managed by the methods of `malloc()`, `calloc()`, `realloc()`, and `free()`.

The variables in section `.data` and `.bss` are arranged and initialized by their assigned value or default value at compile time. The variables in heap and stack are arranged according to run-time state. Note that the order of variables on the stack is not the same as the order of those variables in the source code. When the size of the free memory block in heap satisfies the invocation, the memory area will be allocated to the process. After the allocation, the memory is marked as reserved and a pointer to this location is returned.

In general, global, stack, and heap regions are modeled in different ways. The layout of the global region is decided by variables in `.data` and `.bss`. For stack region, the layout inside a called stack frame is determined by the compiler. Local variables leverage a memory alignment strategy generally. However, the memory allocation for local variables within branches (i.e., *if*, *for*, and *while*) varies with different compilers and optimization levels. Some compilers will optimize the memory allocation of local variables. Therefore, the address allocation of variables within a frame needs to be analyzed by the ELF file. Based on the assumption of an acyclic call, we can analyze the relative order of call propagation from the information in the CFG. Then, combining the function stack frames calling order and the internal information of the frames,

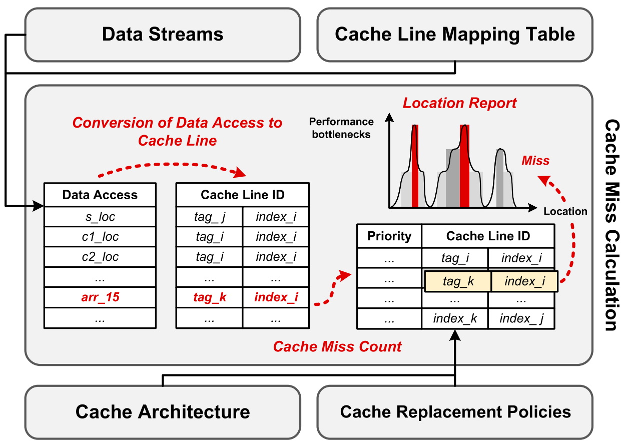


Fig. 7. Cache miss calculation gets the cache line access sequence according to data streams and the cache line mapping table. Then, it simulates cache architecture and cache replacement policies to count the cache misses on execution paths.

we obtain the variable allocation model of the stack region. The memory declared by `malloc()` is stored in the heap region, and the function returns a pointer to the starting address of the allocated memory. The information on array size is lost in the compiler IR optimization, so we get the size from the compiler front-end information. The heap region is managed by the programmer and can be freely allocated at runtime, where resources are requested and released depending on the runtime state. We consider the case where the heap region memory is requested in a two-phase locking way, that is, the memory can only be free after all `malloc()` are requested. The two-phase locking model ensures the continuity of memory blocks in the heap region address and the correction of the analysis.

F. Cache Miss Calculation

The cache miss calculation component supports fully associative and set-associative cache models with various cache replacement policies (e.g., LRU, FIFO, and MRU). As shown in Fig. 7, different cache replacement policies are used in the virtual cache architecture to estimate cache misses. The data access sequence represented by data streams is converted into a cache line access sequence. For the LRU algorithm, the cache miss calculation can be modeled as LRU stack distance to simplify the analysis. In particular, CBANA implements the LRU policy in two ways: one is based on approximate LRU stack distance, and the other is based on the update of priority. The former method provides greater efficiency while the latter method provides greater accuracy. For other policies, each cache line provides a field to record the content saved in the current cache line and sets a state bit to store priority. For example, for any memory access, the content and priority of the cache line in the cache model are updated. If the current variable hits, the priority of the line is updated according to the replacement policy; if there is a cache miss, the strategy further determines whether the set is full and the cache contents need to be replaced, and updates the cache line priority. When a cache

TABLE II
WORKLOADS USED FOR THE EVALUATION OF CBANA AND OTHER ANALYSIS TOOLS

Workload Type	Name	Memory Access Patterns	Number of Memory Accesses	Description
Synthetic	load*_loop*_ir	irregular	$[10^4, 10^9]$	The workload loads data * times in * loop iterations by using the random indexes.
	load*_loop*_r	regular	$[10^4, 10^5]$	The workload loads data * times in * loop iterations by using a consecutive index.
	br_path	regular/irregular	$[10^5, 10^7]$	The workload includes * branches and the executing path is *.
PolyBench [42]	trisolv	regular	10^5	Triangular matrix solver using forward substitution.
	mvt	regular	10^5	Matrix vector multiplication composed of another matrix vector multiplication but with transposed matrix.
	lu	regular	10^5	LU decomposition without pivoting.
	durbin	regular	10^5	Durbing is an algorithm for solving Yule-Walker equations, which is a special case of Toelitz systems.
	jacobi_2d	regular	10^5	Jacobi-style stencil computation over 2D data with 5-point stencil pattern.
	seidel_2d	regular	10^5	Gauss-Seidel style stencil computation over 2D data with 9-point stencil pattern.
Graph [43]	tc	irregular	$[10^5, 10^6]$	Triangle Count (TC) counts the total number of triangles in a graph, and also counts the number of triangles associated with each vertex.
	pr	irregular	10^5	PageRank (PR) algorithm attempts to increase the relative rating of a node based on the weights of all the nodes it is connected to.

miss occurs, we can trace it back to the exact variable and the source code location that caused the cache miss.

IV. Evaluation

In this section, we provide an extensive analysis of the performance of CBANA. Firstly, we describe the experimental setup of the evaluation. Secondly, we evaluate the cache miss analysis of CBANA and Perf for different data regions and various data structures. In addition, we also compare it with the existing representative analysis tools Perf and Valgrind on the commonly used benchmarks PolyBench [42] and graph workloads [43]. Thirdly, we analyze the execution efficiency of CBANA over different kinds of workloads and scales. We compare results among popular dynamic and static analysis tools Perf, Valgrind, Heptane, and our proposed CBANA.

A. Experimental Setup

CBANA is implemented by C/C++ and uses the extracted information of LLVM Clang. Program point and program state are extracted from the program state graph generated by the Clang Static Analyzer (SA). CBANA uses clang-tidy to capture the information. CBANA takes the source code files, related inputs, cache architecture parameters (i.e., cache line size, number of lines, and the associativity of the data cache), and replacement policy as key parameters to model the cache behavior. We use Intel Xeon Silver 4110 @ 2.10GHz as the CPU model for our target machine. It has 32 KB, 8-way set-associative cache (usually corresponding to L1), and the cache line size is 64 Bytes.

Workloads: We perform experiments with 12 types of synthetic workloads with compute/memory intensiveness and various scales, 6 types of workloads from PolyBench [42], and 2 types of workloads from commonly used graph algorithms [43]. Table II lists the workloads. Memory access patterns are classified as either regular or irregular [44]. Regular patterns exhibit sequential access, with a constant stride between consecutive memory addresses. In contrast, irregular patterns have no fixed strides. The number of memory accesses indicates the range of each kind of workload size. For synthetic workloads, some of the large-scale workloads contain several loops and branches to verify the analysis precision and efficiency of CBANA when facing program state explosion. More specifically, workloads are divided into global, stack, and heap regions and various data structures. Other workloads contain selective paths to verify the efficiency of dynamic splicing and the accuracy of path selection. For PolyBench, linear algebra benchmarks (trisolv, mvt, lu, durbin) and stencil benchmarks (jacobi, seidel) use two-dimensional arrays as inputs. The scales of the arrays are default and divided into mini-scale, small-scale, and medium-scale. For the graph workloads, Triangle Count (TC) and PageRank (PR) are implemented with small-scale and medium-scale sizes corresponding to around 100 and 1000 edges in each graph respectively.

Approaches: We compare CBANA with dynamic and static cache miss analysis schemes, namely Perf [9], Valgrind [12], and Heptane [18]. We run the cache miss collector command of Perf on the target machine to get the cache miss results during program execution. To evaluate Valgrind, it is also deployed on the target machine to get the cache miss results. For Heptane, it provides configurable parameters for cache analysis and decouples with the target machine during the time.

Metrics: The metrics are mainly divided into two dimensions: the precision of the analysis results and the efficiency during the analysis. To evaluate the performance of the CBANA, we report the precision of the evaluation results of Perf, Valgrind, Heptane, and CBANA under the same workloads. In addition, we illustrate the analysis time when using different approaches to analyze the cache miss on single and multiple paths.

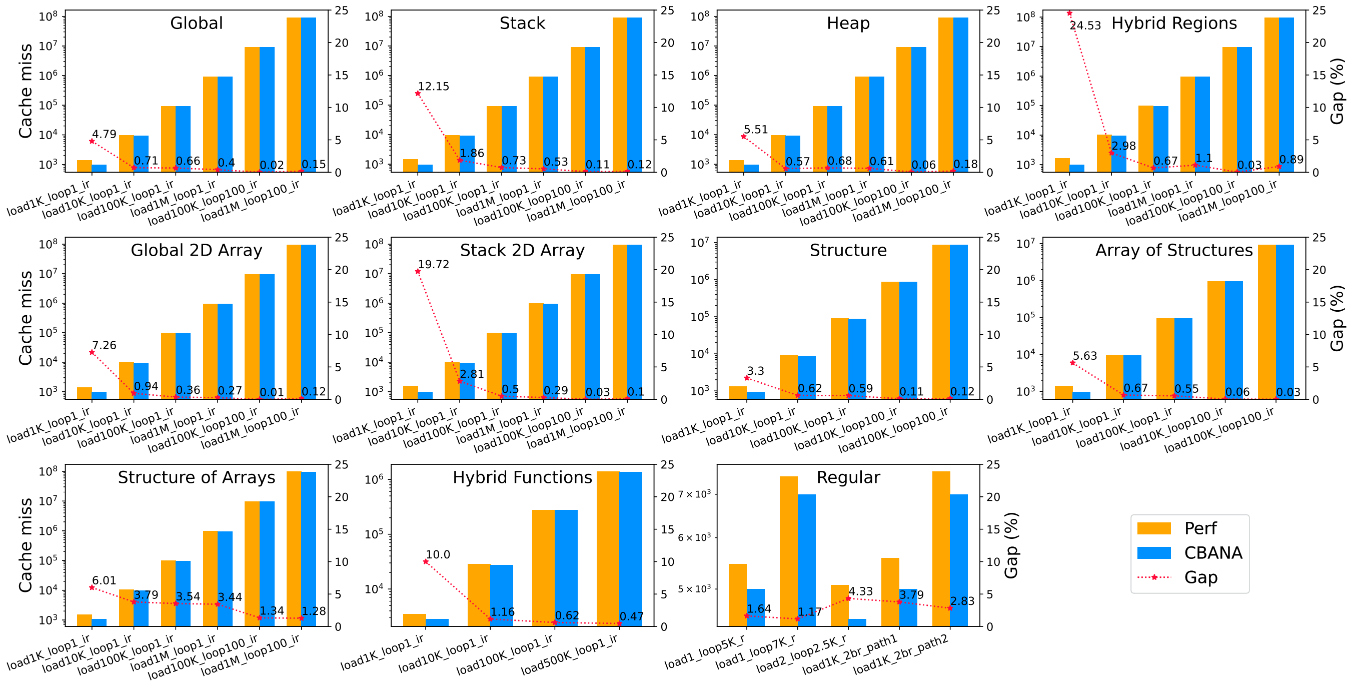


Fig. 8. Comparison of Perf and CBANA for L1 D-Cache miss analysis under different synthetic workloads. Bar plots show the cache miss count. Line plot is the gap (%) between the two bar plots. The gap is highlighted with its exact numeric value. For each workload, the name indicates the number of loops it has and the number of data loaded in each loop. For instance, *load1K_loop1* means there is only a single loop where one thousand data are loaded in each loop, and *load1M_loop1* means one million data are loaded in one loop.

B. Precision of CBANA

We first compare the precision of CBANA with the dynamic analysis method Perf over synthetic workloads. For the Valgrind, the DBI during simulation will lead to a long execution time for certain synthetic workloads, which is impractical in a real scenario. Comparisons with Heptane only include some of the small-scale workloads because large-scale workloads lead to program state explosion and analysis corruption.

CBANA analyzes the data streams of three types of regions, global, stack, and heap regions. In addition, it also analyzes hybrid regions with different scales. As described in Table II, the workloads are identified by the keywords (i.e., *load* and *loop*), which indicate the number of loops included in the code and the number of variables loaded in the loop. The analysis results of cache misses are shown in Fig. 8. The scale of the data loaded in the workloads increases from one thousand to one hundred million. For different regions, global and heap modelings show the most predictable results. For stack region, there is a fluctuation of prediction error when the scale of the data access is small, which indicates that there is still room for improvement in the precision of runtime behavior modeling.

Fig. 8 also illustrates the CBANA's precision of cache miss analysis oriented to common structures, i.e., array, structure, array of structures, and structure of arrays, that may cause the problem of data locality. As the program scale increases, CBANA implements a similar cache miss analysis result to Perf. This indicates the correctness of structure and array modeling in memory address mapping. The fine-grained analysis of CBANA helps analyze the data affinity in data structures, which further benefits the reorganization of the layout in data structures. In addition, the workloads in the regular type include branches. Compared with irregular workloads, regular workloads achieve lower cache miss gaps for the same scale of data accesses. The main reason is extra cache misses caused by the operating system. For instance, irregular memory accesses will introduce more interferences of OS such as the execution of frequent context switch and page/cache line replacement. The results also show that the input-aware path analysis scheme could calculate the number of cache misses on different paths with multiple sets of inputs respectively. For each path, we use a consecutive index to access the array to realize a regular memory access pattern. The overall results show that the gap of CBANA decreases with the expanded scale and eventually remains below 3.79% for ten thousand data accesses.

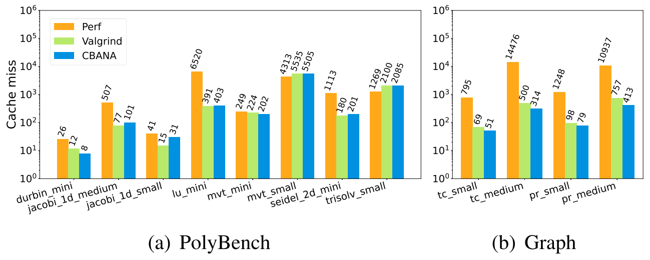


Fig. 9. Comparison of Perf, Valgrind, and CBANA for L1 D-Cache miss analysis in (a) PolyBench and (b) Graph workloads. Bar plots show the cache miss count of different approaches.

Fig. 9 shows the cache miss analysis of both PolyBench and graph workloads, which include nested loops and fixed-step iterating array access. As demonstrated in Fig. 9(a) and 9(b),

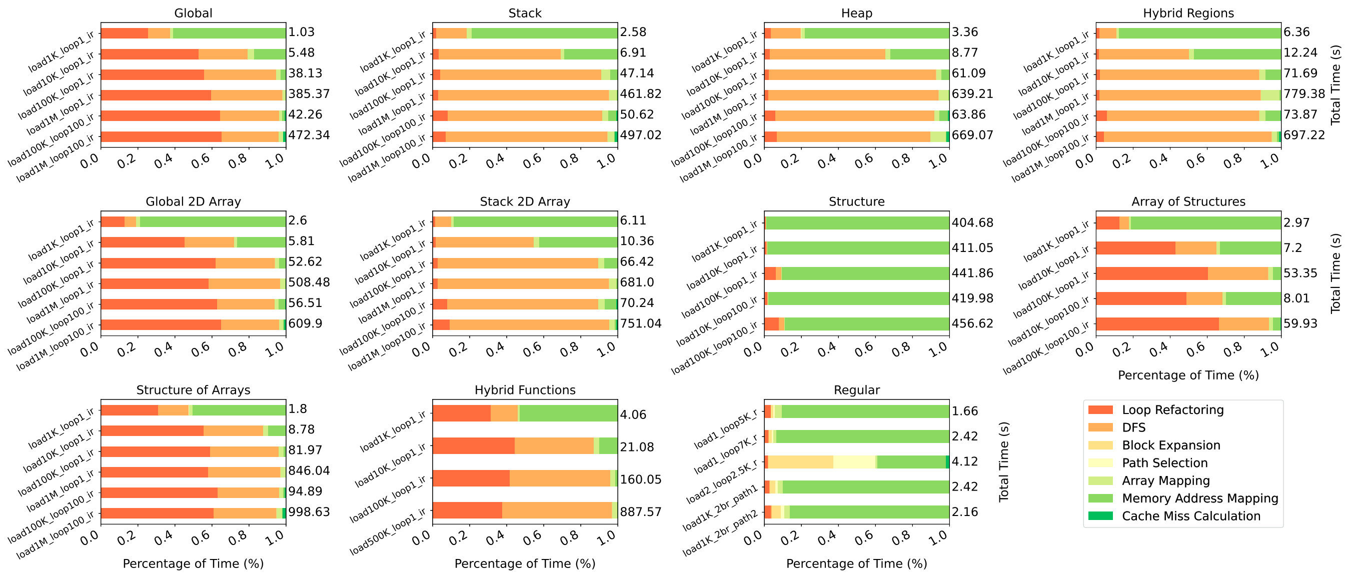


Fig. 10. Breakdown of the total execution time of CBANA over different synthetic workloads.

the results of Valgrind and CBANA have the same analyzing trend. Cache miss gaps between Valgrind and CBANA are introduced by library functions that are dynamically loaded and have different behaviors based on the program itself, resulting in higher cache miss results (e.g., tc, pr, etc) for Valgrind. However, compared with the dynamic analysis tool Perf, the cache miss results of some workloads (e.g., jacobi, lu, seidel, etc) vary enormously. The reason could be the extra cache misses caused by operating systems and library functions, even though the program is running individually. Compared with CBANA, we analyze that the lower results (e.g., mvt and trisolv) of Perf are caused by hardware optimization (i.e., hardware prefetching for first-level data cache [45]). However, CBANA only simulates the cache behavior, which is a limitation of static analysis tools in general. In particular, it is found that the cache miss profiling results increased linearly with the runtime of the program when `sleep()` was used in the code. This situation may lead to errors in the results of dynamic analysis. Cache misses introduced by operating system cannot be analyzed by simulation or static estimation. Nevertheless, this gap will be bridged as the scale of the program increases.

C. *Efficiency of CBANA*

The execution overhead of CBANA is shown in Fig. 10, which can be generally divided into preprocessing for the whole program and cache miss analysis for each execution path. For the preprocessing, it includes: (1) flow sensitive and path sensitive analysis shown as the time of loop refactoring, DFS, and block expansion, and (2) data mapping analysis shown as the time of array mapping and memory address mapping. For the cache miss analysis, it includes input-aware path analysis, dynamic splicing of intermediate results, and cache miss calculation. The former is shown as the time of path selection in the breakdown of the total execution time. The compilation time is not listed, as the time is little and depends on the compiler. As the scale of data accesses grows, the time of loop refactoring is increased. For small-scale workloads, memory address mapping takes up a major portion of the time overhead. As the scale of the workloads increases, loop refactoring and DFS take up most of the time overhead and memory address mapping becomes less of a bottleneck. In fact, the time of loop refactoring and DFS is influenced by the growing number of paths in the exploded graph. More specifically, the amount of data inside the loop affects the loop refactoring time mainly, while the number of loops affects the DFS time more. Memory address mapping occupies the most time in small-scale workloads.

TABLE III
COMPARISON OF THE TIME OF PROGRAM STATE ANALYSIS BEFORE AND AFTER LOOP REFACTORING

Workload Name	Original		Loop Refactoring	
	Time (s)	Number of nodes	Time (s)	Number of nodes
load10K_loop100_ir	35.377	11006022	2.628	110089
load100K_loop100_ir	405.094	110033022	26.123	1100359

Table III shows the difference in program state analysis before and after the loop refactoring in dynamic splicing. The original time is collected during the period when Clang SA analyzes the total nodes in a program state graph. After loop refactoring, the number of nodes in the program state graph will be greatly reduced. The analysis time and the number of nodes are mainly related to the number of loop iterations and the number of program states in the loop, so the optimization effect of different programs is different. Our scheme reduced time overhead by more than 92.57% in loop analysis. Overall, CBANA is the first static analysis tool to address state explosion in large-scale programs.

TABLE IV
EFFICIENCY MEASUREMENT (SINGLE PATH)

Workload Type	Workload Name	Dynamic Analysis				Static Analysis			
		Perf [9]		Valgrind [12]		Heptane [18]		CBANA	
		Cache Miss	Total Time (s)	Cache Miss	Total Time (s)	Cache Miss	Total Time (s)	Cache Miss	Total Time (s)
Synthetic	load600_loop50_r	30144	0.29	30047	0.36	30001	97.59	29225	3.71
	load800_loop50_r	40280	0.29	40047	0.37	40001	171.10	40001	4.92
	load1K_loop50_r	50249	0.30	50042	0.37	50001	267.23	50001	6.12
PolyBench [42]	durbin_mini	26	0.25	12	0.53	-	-	8	0.59
	jacobi_1d_medium	507	0.26	77	4.82	-	-	101	70.92
	jacobi_1d_small	41	0.26	15	1.38	-	-	31	3.53
	lu_mini	6520	0.27	391	2.64	-	-	403	83.72
	mvt_mini	249	0.27	224	0.74	-	-	202	10.09
	mvt_small	4313	0.27	5535	3.48	-	-	5505	1073.43
	seidel_2d_mini	1113	0.27	180	4.09	-	-	201	139.16
	trisolv_small	1269	0.27	2100	1.20	-	-	2085	94.66
Graph [43]	tc_small	795	0.27	69	0.80	-	-	51	7.05
	tc_medium	14476	0.29	500	7.15	-	-	314	703.16
	pr_small	1248	0.27	98	0.78	-	-	79	4.47
	pr_medium	10937	0.28	757	3.22	-	-	413	169.73

Note: The symbol “-” indicates the data can not be obtained within 30 minutes.

TABLE V
EFFICIENCY MEASUREMENT (MULTIPLE PATHS)

Workload Name	Execution Path	Dynamic Analysis				Static Analysis			
		Perf [9]		Valgrind [12]		CBANA			
		Cache Miss	Total Time (s)	Cache Miss	Total Time (s)	Cache Miss	Preprocessing Time (s)	Cache Miss Analysis Time (s)	Total Time (s)
load1K_2br_	path1	509190	272.46	501391	832.48	500001	6.32	14.42	20.75
	path2	14356		6391		5001			
load1K_3br_	path1	5011531	287.22	5001398	1617.85	5000102	6.32	239.36	245.67
	path2	59513		51398		50011			
	path3	9873		2398		1002			

Table IV demonstrates the comparisons of cache miss and analysis time among Perf, Valgrind, Heptane, and CBANA over the workloads with a single path. Synthetic workloads are designed to evaluate the cache miss gaps with large scales of data accesses, which reduces the influence of cache misses caused by the operating system in dynamic analysis methods. Compared with the dynamic analysis tools Perf and Valgrind, CBANA achieves maximum cache miss gaps of 3.05% and 2.74% respectively for synthetic workloads. Compared with the static analysis tool Heptane, CBANA reduces analysis time by up to 97.71%. Furthermore, Table IV shows that our approach can get results those cannot be obtained by Heptane within an appropriate large time limit for PolyBench and graph workloads.

Though the execution time of CBANA is not superior to dynamic approaches for tackling inputs with a single path, when considering multiple paths, our method is much better. The results are summarized in Table V. For each workload, we randomly generate 1000 sets of inputs for the multiple-path workloads (i.e., *load1K_2br_path** and *load1K_3br_path**) and get the cache miss analysis results for each path. Compared with Perf and Valgrind, the total analysis time can be reduced by up to 92.38% and 97.51% separately. The reason is that CBANA analyzes the whole program during preprocessing, and it can leverage the preprocessed intermediate results to reduce computation for multiple paths. Therefore, as the number of input sets increases, CBANA shows superior performances.

D. Summary of Evaluation

We compare the precision and efficiency of CBANA with other analysis approaches in different benchmarks. Our results show that CBANA achieves the same precision as other static methods and is also capable of analyzing large-scale programs precisely. When compared with the dynamic analysis tool Perf, the cache miss gap decreases with the increase in the number of data accesses and is less than 3.79% for ten thousand data accesses. The overall gap rate can be even less than 0.89% when the number of data accesses is large. Compared with Valgrind, the cache miss gap is less than 2.74%. For multiple-path workloads, CBANA achieves up to 92.38% and 97.51% time reduction compared with Perf and Valgrind, respectively. Compared with the popular static cache analysis tool Heptane, it achieves a time reduction of 97.71% when analyzing single-path workloads.

V. CONCLUSION

In this study, we describe the design and implementation of a lightweight, efficient, and flexible cache behavior analysis framework, CBANA. With the detailed information extracted from the source level, CBANA provides input-aware path analysis that ensures the accuracy of branch analysis in static methods. The module of dynamic splicing of intermediate results mitigates the risk of program state explosion in static analysis approaches. In addition, the decoupling of the framework establishes the flexibility of CBANA, which supports a variety of alternative cache replacement policies. We implement CBANA and evaluate it on synthetic, PolyBench, and graph workloads. The experimental results show that CBANA can detect cache misses efficiently and flexibly with limited static information. Compared with the analysis result of the program running state collected by Perf, the gap of CBANA is less than 3.79% for ten thousand data accesses and can be less than 0.89% with a larger number of data accesses. Compared with the dynamic analysis tool Valgrind, CBANA offers close analysis accuracy and provides configurable parameters for the memory architecture of the target machine. When compared with the widely-used dynamic analysis tools Perf and Valgrind, for the synthetic workloads with over ten thousand data accesses, the cache miss gap is below 3.79% and 2.74% respectively. Additionally, it provides time reduction up to 92.38% and 97.51% for multiple-path workloads. CBANA also provides 97.71% time reduction compared with the widely used static analysis tool Heptane.

In the future, we can explore more complex situations by further considering caching methods (i.e., cache write policy and write allocation policy) and more system kernel information (i.e., program input, the number of threads, and thread scheduling strategy) to support cache behavior analysis for L2 and L3 caches and interaction between different threads and cores.

Acknowledgment

The authors would like to thank anonymous reviewers for their constructive comments.

References

[1] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "A modern primer on processing in memory," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Singapore: Springer Nature Singapore, 2022, pp. 171–243.

[2] A. Shrestha, H. Fang, Z. Mei, D. P. Rider, Q. Wu, and Q. Qiu, "A survey on neuromorphic computing: Models and hardware," *IEEE Circuits Syst. Mag.*, vol. 22, no. 2, pp. 6–35, 2nd Quart. 2022.

[3] I. Cutress and A. Frumusanu, "The intel 12th Gen core i9-12900K review: Hybrid performance brings hybrid complexity." AnandTech. [Online]. Available: https://www.anandtech.com

[4] T. L. Johnson and W.-M. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 315–326, 1997.

[5] O. Ozturk, U. Orhan, W. Ding, P. Yedlapalli, and M. T. Kandemir, "Cache hierarchy-aware query mapping on emerging multicore architectures," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 403–415, Mar. 2017.

[6] M. Badamo, J. Casarona, M. Zhao, and D. Yeung, "Identifying power-efficient multicore cache hierarchies via reuse distance analysis," *ACM Trans. Comput. Syst.*, vol. 34, no. 1, pp. 1–30, 2016.

[7] S. Bijo, E. B. Johnsen, K. I. Pun, and S. L. T. Tarifa, "A formal model of data access for multicore architectures with multilevel caches," *Sci. Comput. Program.*, vol. 179, pp. 24–53, Jun. 2019.

[8] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Dept. Defense HPCMP Users Group Conf.*, vol. 710, 1999, pp. 1–8.

[9] "perf: Linux profiling with performance counters." Linux. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[10] C. Yu, P. Roy, Y. Bai, H. Yang, and X. Liu, "LWPTool: A lightweight profiler to guide data layout optimization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2489–2502, Nov. 2018.

[11] F. Bellard, "QEMU, a fast and portable dynamic translator." in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[12] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[13] T. Van Dung, I. Taniguchi, and H. Tomiyama, "Cache simulation for instruction set simulator QEMU," in *Proc. IEEE 12th Int. Conf. Dependable, Autonomic Secure Comput.,* Piscataway, NJ, USA: IEEE Press, 2014, pp. 441–446.

[14] C. Sung, B. Paulsen, and C. Wang, "CANAL: A cache timing analysis framework via LLVM transformation," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 904–907.

[15] H. Brais, R. Kalayappan, and P. R. Panda, "A survey of cache simulators," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–32, 2020.

[16] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1977, pp. 238–252.

[17] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 56–67, 2007.

[18] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *Proc. 17th Int. Workshop Worst-Case Execution Time Anal. (WCET),* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017, pp. 8:1–8:12.

[19] D. Chen, F. Liu, C. Ding, and S. Pai, "Locality analysis through static parallel sampling," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 557–570, 2018.

[20] M. Wu and C. Wang, "Abstract interpretation under speculative execution," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2019, pp. 802–815.

[21] A. Adileh, D. J. Lilja, and L. Eeckhout, "Architectural support for probabilistic branches," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Piscataway, NJ, USA: IEEE Press, 2018, pp. 108–120.

[22] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv. (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[23] Q. Wang, X. Liu, and M. Chabbi, "Featherlight reuse-distance measurement," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 440–453.

[24] G. Stock, S. Hahn, and J. Reineke, "Cache persistence analysis: Finally exact," in *Proc. IEEE Real-Time Syst. Symp. (RTSS),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 481–494.

[25] X. Liu, K. Sharma, and J. Mellor-Crummey, "ArrayTool: A lightweight profiler to guide array regrouping," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 405–415.

[26] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "DMon: Efficient detection and correction of data locality problems using selective profiling," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2021, pp. 163–181.

[27] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO),* Piscataway, NJ, USA: IEEE Press, 2018, pp. 613–626.

[28] S. S. Banerjee, S. Jha, Z. Kalbarczyk, and R. K. Iyer, "BayesPerf: Minimizing performance monitoring errors using Bayesian statistics," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 832–844.

[29] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?" in *Proc. Asia Conf. Comput. Commun. Secur.*, 2018, pp. 457–468.

[30] D. Hardy and I. Puaut, "Predictable code and data paging for real time systems," in *Proc. Euromicro Conf. Real-Time Syst.,* Piscataway, NJ, USA: IEEE Press, 2008, pp. 266–275.

[31] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim., (CGO),* Piscataway, NJ, USA: IEEE Press, 2004, pp. 75–86.

[32] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2003, pp. 245–257.

[33] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proc. 17th Annu. Int. Conf. Supercomputing*, 2003, pp. 150–159.

[34] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, pp. 1–39, 2009.

[35] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. & Softw. (ISPASS)*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 55–65.

[36] M. A. Sasongko, M. Chabbi, M. B. Marzijarani, and D. Unat, "Reuse-Tracker: Fast yet accurate multicore reuse distance analyzer," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, pp. 1–25, 2021.

[37] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[38] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," *ACM SIGPLAN Notices*, vol. 39, no. 6, pp. 255–266, 2004.

[39] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. {USENIX} Annu. Tech. Conf. ({USENIX} {ATC} 16)*, 2016, pp. 351–364.

[40] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 2526–2540.

[41] R. E. Bryant, O. David Richard, and O. David Richard, *Computer Systems: A Programmer's Perspective,* vol. 2 Englewood Cliffs, NJ, USA: Prentice Hall, 2003.

[42] T. Yuki and L.-N. Pouchet, "Polybench/c 4.2." SourceForge. [Online]. Available: https://sourceforge.net/projects/polybench/

[43] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.

[44] T. Hussain, "A novel access pattern-based multi-core memory architecture," Accessed: Nov. 11, 2022.

[45] I. Coorporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2023. [Online]. Available: https://cdrdv2-public.intel.com/671488/248966_software_optimization_manual-1.pdf

Qilin Hu received the M.S. degree in computer technology from Hunan University, China, in 2021. She is currently working toward the Ph.D. degree in computer science and technology with Hunan University, China. Her research interests include parallel and distributed computing, high-performance computing, and system performance analysis.

Yan Ding (Member, IEEE) received the Ph.D. degree in computer science from Hunan University, China, in 2021. He is currently an Assistant Professor with Hunan University. His research interests include parallel computing, mobile edge computing, big data, artificial intelligence, and architecture. He has published 8 papers in journals and conferences, including Design Automation Conference, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Services Computing, IEEE Transactions on Industrial Informatics, *Journal of Parallel and Distributed Computing, Computers & Security*, and 17th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2019). He received the Outstanding Paper Award in the 17th IEEE ISPA.

Chubo Liu (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and technology from Hunan University, China, in 2011 and 2016, respectively. He is currently a Full Professor of computer science and technology with Hunan University. His research interests include parallel and distributed computing, computer architecture, artificial intelligence, game theory, and approximation and randomized algorithms. He has published over 40 papers in journals and conferences such as the IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Cloud Computing, IEEE Transactions on Mobile Computing, IEEE Transactions on Industrial Informatics, IEEE Internet of Things Journal, *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, *Theoretical Computer Science*, ISCA, DAC, and NPC. He won the IEEE TCSC Early Career Researcher (ECR) Award in 2019. He is a Member of ACM.

Keqin Li (Fellow, IEEE) received the B.S. degree in computer science from Tsinghua University, in 1985 and the Ph.D. degree in computer science from the University of Houston, in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University, China. He has authored or co-authored more than 1000 journal articles, book chapters, and refereed conference papers. He holds nearly 75 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top five most influential scientists in parallel and distributed computing in terms of single-year and career-long impacts based on a composite indicator of the Scopus citation database. He is an AAAS Fellow, an AAIA Fellow, and an ACIS Founding Fellow. He is a Member of Academia Europaea (Academician of the Academy of Europe).

Kenli Li (Senior Member, IEEE) received the M.S. degree in mathematics from the Central South University, China, in 2000, and the Ph.D. degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a Visiting Scholar with the University of Illinois, Urbana-Champaign, IL, USA from 2004 to 2005. He is a Full Professor of computer science and technology with Hunan University. His research interests include parallel and distributed processing, supercomputing and cloud computing, high-performance computing for Big Data and artificial intelligence, etc. He has published more than 300 papers in international conferences and journals. He is currently served on the editorial boards for IEEE Transactions on Computers. He is an outstanding Member of CCF.

Albert Y. Zomaya (Fellow, IEEE) is the Peter Nicol Russell Chair Professor of computer science with the School of Computer Science, Sydney University, and serves as the Director of the Centre for Distributed and High-Performance Computing. He has published more than 700 scientific papers and articles and is author, co-author or editor of more than 30 books. He is the Editor-in-Chief of the *ACM Computing Surveys* and serves as an Associate Editor for several leading journals. He is a Decorated Scholar with numerous accolades including Fellowship of the AAAS and the IET. Also, he is a Fellow of the Australian Academy of Science, a Fellow of the Royal Society of New South Wales, a Foreign Member of Academia Europaea, and a Member of the European Academy of Sciences and Arts. His research interests include parallel and distributed computing, networking, and complex systems.