An Empirical Study: mems as a Static Performance Metric

Liwei Zhang[1,2,3], Baoquan Cui[2,3], Xutong Ma[4], Jian Zhang[1,2,3,*]
[1]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
[2]Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences (CAS), Beijing, China
[3]University of Chinese Academy of Sciences (UCAS), Beijing, China
[4]Inria, Paris, France
{zhanglw, cuibq, zj}@ios.ac.cn, xutong.ma@inria.fr
*Corresponding author

Abstract—Performance analysis is essential to ensure the non-functional performance requirements of a software system. However, existing runtime-based approaches suffer from the issues of efficiency and platform dependency. In this paper, we investigate the effectiveness of using the *mems* value to statically estimate the program performance. The *mems* value, originally proposed by Donald Knuth, is a static and architecture-independent metric used to measure memory access, to estimate program performance statically. We developed an instrumentation tool to record the control flow and measure the *mems* value by rewriting the source code. Experimental results across ten classical algorithm programs show that execution paths of a program with larger *mems* values consistently exhibit lower efficiency. Whereas the correlation weakens among different programs. This indicates that the *mems* maric is best suited for comparing the performance of various paths in the same program.

Keywords–Static Analysis; Program Performance; Memory Accesses; Performance Metrics

1. Introduction

An important aspect of software quality is performance. Program efficiency is typically characterized by complexity measures such as worst-case or average-case time complexity. However, for real-world programs, even domain experts can find it difficult to accurately predict performance. For example, although the Boyer–Moore string matching algorithm was introduced in 1977 [2], its exact complexity was not rigorously established until more than a decade later [3]. Moreover, theoretical complexity often fails to reflect actual program performance, as it abstracts away constant factors, low-level implementation details, and hardware-specific effects. Therefore, measuring performance through empirical evaluation remains necessary for understanding real-world program behavior.

However, empirical performance evaluation often suffers from limited path coverage, platform dependence, and high measurement cost. These limitations hinder the ability to detect performance regressions and ensure consistent behavior, which are essential aspects of software quality assurance.

To address the gap between theoretical models and practical behavior, researchers have explored static and symbolic analysis methods. A notable example is Zhang's framework for *performance estimation using symbolic data* [1], which uses symbolic execution and volume computation [20, 21] to estimate performance without running the program. The key idea is to represent execution conditions symbolically and compute the volume of input space satisfying each path, enabling quantitative analysis of potential overheads.

Although prior work has explored symbolic and static techniques for program analysis, practical methods for static performance estimation remain underdeveloped. Empirical approaches, while accurate, require repeated execution on target hardware and are difficult to scale across environments. Symbolic execution tools struggle with scalability and automation, particularly for real-world C programs. Most existing static performance metrics—such as operation counts, loop nesting depth, or control-flow complexity—lack empirical validation and have limited ability to explain performance differences between execution paths. In particular, the *mems* metric, originally proposed by Knuth [6] to measure memory access operations, has seen little empirical study or integration into automated static analysis workflows. As a result, developers currently lack lightweight, architecture-independent metrics that can reliably predict path-level performance behavior.

Traditional performance profiling relies on repeated program executions with various inputs to gather timing statistics [17–19]. While these techniques are valuable, they may miss rare or worst-case paths and often require significant computational effort. Automated testing frameworks like *XSTRESSOR* [17] and hybrid tools like those from Bundala and Seshia [18] improve test input quality but still struggle with scalability and low-level performance modeling. To improve generalizability and efficiency, symbolic benchmarking [1] emerged—leveraging symbolic execution to explore multiple paths in a single analysis and estimate performance using metrics such as comparison count or memory accesses.

We adopt the *memory access count*—abbreviated as *mems*—as a machine-independent performance metric. Originally proposed by Knuth [6], *mems* represents the number of memory read/write operations in a program and offers a portable estimate of performance. Unlike raw execution time, which is affected by CPU and system environment, *mems* is a symbolic metric that reflects a program's inherent memory behavior.

Extending this idea, we design a path-sensitive static analysis that uses symbolic execution to compute memory accesses for each feasible control-flow path. By applying model counting techniques [20, 21], we compute the expected performance as a weighted average over path-level mems. This method enables early-stage and architecture-neutral performance reasoning without requiring actual execution.

Although *mems* has been discussed in theoretical contexts [1, 6], its practical applicability in real-world software analysis remains unexplored. Several technical challenges must be addressed to realize its potential. First, memory access patterns in C programs are often implicit and context-sensitive, making it difficult to statically and precisely extract *mems* without full semantic understanding. Second, the relationship between static memory access counts and dynamic performance is nontrivial: it may vary across paths, compilers, and hardware configurations, raising questions about when and where *mems* is a valid predictor. Third, applying such analysis to large codebases involves scaling across thousands of execution paths while preserving both soundness and efficiency. These challenges underscore the need for systematic tools and empirical validation to assess *mems* as a practical performance metric.

To address these challenges, we propose and implement the following:

- **New Metric.** We adopt *mems*, originally proposed by Knuth, as a lightweight and architecture-independent performance metric based on static memory access counts, with a particular focus on array operations as key contributors to execution cost.
- **Automated Solution.** We develop an automated pipeline for *mems* analysis based on the `eppather` tool [22], derived from the `epat` test generation framework [25, 26]. Our tool integrates AST-based analysis, path traversal, and source-to-source instrumentation to collect *mems* and timing data.
- **Evaluation.** We empirically evaluate whether *mems* correlates with execution time at two levels: (1) across different paths in the same program (RQ1), and (2) across programs and platforms (RQ2). Results show that *mems* reliably predicts intra-program performance, but its cross-program accuracy varies.

The findings of this empirical research contribute directly to improving the accuracy and reliability of static performance indicators used in symbolic benchmarking. By clarifying the strength and limitations of *mems* as a performance predictor, this study does not only enhance the understanding of program complexity, but also provides practical guidance for more effective performance evaluation, optimization strategies, and software testing practices.

2. Preliminary

Static program analysis plays a vital role in program understanding, optimization, and verification. One commonly used representation is the control flow graph (CFG), where nodes represent statements or basic blocks, and edges represent possible control flow between them. Many static analysis techniques, such as abstract interpretation and symbolic execution, operate on CFGs to detect bugs, verify properties, or estimate performance.

2.1 The *mems* Metric

Usually we evaluate a program's performance by executing it on a computer with input data, and measure its duration of execution. This can be done repeatedly with different input data. As abstracted as modeled with indicated in Moore's Law, the improvement in computer devices leads to an increment in the efficiency of program execution as time passes. For many years, computers have been getting faster and faster. Thus a program's running times (in seconds) are getting less and less.

Knuth came up with an idea to measure a program's performance, which is independent of the processor's computing power. He proposed to use the number of memory accesses (in short, *mems*) [6] to indicate the execution speed of a program.Knuth has evaluated various SAT solvers using the *mems* metric [27]. We simply introduce a special variable – *mems*, which is initialized to 0 and incremented every time a memory access (such as an array read or write) occurs during the execution. In a statement which involves memory access, we add a statement to increase the value of *mems*. Knuth often uses the notation *o* to increase the value by 1, *oo* to increase the value by 2, and so on. Thus we may have statements like the following, to compute the value of *mems*.

```
 o, arr[i] = i+2;
oo, arr[i+1] = arr[i];
```

The *mems* has not been used widely. In the subsequent sections of this paper, we will conduct a series of experiments to check whether this indicator is feasible.

2.2 Path-Based Performance Estimation

A widely adopted method in static analysis is *path-sensitive analysis*, where a program is analyzed along individual execution paths extracted from the CFG (Control Flow Graph). Each path begins at the program's entry and ends at an exit point. Analyzing individual paths allows for fine-grained understanding of the program's behavior.

However, due to the condition checks in a program, not all topographically valid paths extracted from a CFG are feasible. A path is considered feasible only if its path constrints (aka. path conditions), the conjunction of all its conditional clauses over the program inputs, are satisfiable. To determine the feasibility, we can first use the symbolic execution approach to extract the path constraints for a given path, and then check the satisfiability [10, 11, 20] with SMT solvers or compute the path frequency, the number of input assignments satisfying the path constraints, with model counters.

To estimate performance in a machine-independent manner, we define *Performance Indicators* (PINDs), such as the number of memory accesses, arithmetic operations, or comparisons on a given path. Let a program consist of a set of paths $\{P_i\}$. Each path P_i has [9]:

- A frequency δ_i — how often the path is executed (based on the number of satisfying inputs).

- A PIND value pind_i — the cost incurred along the path (e.g., number of memory operations).

The estimated performance of the entire program is then computed as the weighted average:

$$\text{Performance} = \frac{\sum_i (\delta_i * \text{pind}_i)}{\sum_i \delta_i}$$

This technique has been extended to characterize not only average cost but the full distribution of performance across the input space. For example, Chen et al. [28] proposed an approach using probabilistic symbolic execution to derive *performance distributions* rather than single-point estimates. This enables modeling not just expected runtime, but also the variance and tail behaviors of performance under uncertainty.

To represent conditional branches along an execution path, we use the notation `@(condition)` to denote the occurrence of a branching condition within the code, i.e. the pre-conditions of this path. This can be done via symbolic execution [5].

2.2.1 Example

Consider the following code snippet:

```
int x;
@( (x > 20) && (x <= 100) )
x = x - 10; if (x > 30) {...} else {...}
```

This format is also used in our instrumentation output to log the sequence of conditions encountered during execution. This program has two feasible paths:

- P_T: takes the branch where `x - 10 > 30`
- P_F: takes the branch where `x - 10 <= 30`

The corresponding path conditions are:

```
P_T: (x > 20) && (x <= 100) && (x - 10 > 30)
P_F: (x > 20) && (x <= 100) && (x - 10 <= 30)
```

With a model counting tool [4, 7, 21], we will have: $\delta_T = 60$ and $\delta_F = 20$. For simplicity, we assume the following *mems* values for the two branches as hypothetical examples: $\text{pind}_T = 3$ and $\text{pind}_F = 2$. Then the overall performance estimate is:

$$\text{Performance} = \frac{60 * 3 + 20 * 2}{60 + 20} = \frac{260}{80} = 2.75$$

This value reflects the average number of operations to be executed with any concrete inputs in the input domain.

2.2.2 Remarks

This method is particularly useful in early-stage design and static optimization, where empirical timing information is not yet available. It can also guide compiler decisions or resource estimation in embedded systems, where performance metrics like memory operations are more meaningful than raw execution time.

3. Approach

The concept of using memory access counts, *mems*, as a machine-independent performance indicator was first proposed by Knuth. In his book *The Stanford GraphBase* [6] and further discussed in the *The Art of Computer Programming* [16], Knuth introduced the idea of instrumenting programs with a *mems* counter to reflect the computational cost associated with memory operations, independent of processor speed.

Building on this idea, our previous work [1] proposed a symbolic estimation framework in which *mems* is calculated along individual control-flow paths. The paper further suggested aggregating these per-path *mems* values using weighted averages based on symbolic path frequencies to obtain a performance estimate for the entire program.

However, despite the elegance of this theory, none of these studies provided empirical validation on real-world programs. As a result, it remains unclear whether *mems* values meaningfully correlate with execution time in practice, especially for modern C programs across various input paths and platforms.

To bridge this gap, we perform large-scale experiments by executing instrumented C programs and collecting path-level metrics—*mems*, path length, and runtime. This requires the construction of a precise and automated instrumentation framework, detailed below.

3.1 Instrumentation Pipeline

To facilitate data collection, we built a Clang-based tool named `eppather-clangpass`. This tool integrates directly with Clang's AST infrastructure and modifies the source code by inserting instrumentation logic into user-defined functions (excluding `main`). During execution, the instrumented program records the control-flow path, memory access count (*mems*), path length, and timing metrics, all printed to standard output for batch analysis. The instrumentation pipeline operates in the following stages:

- **Function Entry and Exit.** At the entry of each user-defined function, we insert definitions and initializations for the variables used to track performance metrics, including *mems*, `path_len`, and timing variables such as `start`, `end`, and `freq` (on Windows, using `QueryPerformanceCounter` APIs). At the function exit (before every `return` statement), we insert logic to compute and print the total execution time, memory access count, and path length.
- **Conditional and Loop Statements.** For every conditional or loop predicate (e.g., `if`, `while`, `for`), we inject a `printf` statement to record the evaluation of the branch condition. The true branch is annotated as `@(condition)`, while the false branch is annotated as `@(!(condition))`, where `condition` is the string representation of the original source predicate. Each branch point also increments the path length counter.
- **Assignment Statements.** Each assignment operation is instrumented with a `printf` that logs the source-level statement as a step in the execution path. This captures the concrete path taken through the program logic.

- **Array Access Detection.** For assignments or expressions involving array reads or writes (e.g., `arr[i] = ...` or `... = arr[i]`), we increment the *mems* counter accordingly. For example, a statement like `arr[i] = arr[i] + 1` is counted as two memory operations— read and write.

This instrumentation strategy allows us to track symbolic path conditions, memory usage, and runtime characteristics in a unified and automated way. The resulting output can then be parsed for further batch analysis to assess the relationship between static metrics (like *mems*) and dynamic behavior (like execution time).

3.2 Instrumentation Example

The following listing shows a simplified instrumented version of a loop-based test program used in RQ1. *Red-highlighted lines* represent the original logic of the program (e.g., control structures and memory operations), while the *non-highlighted lines* correspond to the inserted instrumentation code. Instrumentation logic is placed before and after control branches and memory-related expressions to ensure precise path tracking, memory access counting, and execution time measurement.

Listing 1. Simplified instrumented version with original code highlighted

```
LARGE_INTEGER freq, start, end;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start);
int a = 0, b = 0, i;
for (i = 0; i < n; i++) {
   path_len = path_len + 1;
   if (mode > 0) {
      path_len = path_len + 1;
      arr[i] = i * 2 + arr[i];
      mems = mems + 2;
      mode = mode - 1;
   } else {
      path_len = path_len + 1;
      b = i * 3 + b;
      mode = mode - 1;
   }
}
printf("Total path length: %d\n", path_len);
printf("Total memory accesses: %d\n", mems);
QueryPerformanceCounter(&end);
double time_taken = (double)(end.QuadPart -
     start.QuadPart) / freq.QuadPart;
printf("Execution time: %f\n", time_taken);
```

This instrumented version tracks control-flow conditions using annotated `printf` statements, counts array memory accesses using the *mems* counter, and measures execution time using high-precision timers.

4. Evaluation

We evaluate the practicality and predictive power of using memory access counts (*mems*) as a static performance metric through empirical experiments guided by the two research questions outlined above. Each question is addressed via multiple targeted validations as follows:

RQ1: Intra-Program Correlation Between *mems* and Execution Time. This part of the evaluation aims to determine whether *mems* values can effectively indicate performance differences among paths within the same program.

- **Validation 1: Equal-Length Paths.** For paths of equal length within the same program (i.e., same number of control-flow decisions), we examine whether higher *mems* values consistently result in longer execution times. A motivating example program is constructed with a loop and branching logic where some paths involve memory accesses while others do not.
- **Validation 2: Path-Wide Correlation.** We collect data for all feasible execution paths in a program and compute statistical correlations (e.g., Pearson correlation coefficient) between *mems* values and corresponding execution times. This evaluates whether *mems* provides a monotonic signal for performance under realistic control-flow variations.

RQ2: Cross-Program Generalizability of *mems* as a Performance Metric. This part investigates whether *mems* remains a useful indicator of performance across multiple programs and environments.

- **Validation 1: Server-Side Analysis.** We execute a set of classic algorithm implementations on a high-performance Linux server, measuring runtime using both the system `time` command and Valgrind profiling. We compare *mems* values with observed execution times across different programs.
- **Validation 2: Optimization-Free Local Execution.** To reduce noise from compiler optimizations, we rerun all tests on a local Windows machine with optimization flags disabled (`-O0`). Execution time is recorded using high-precision timing APIs, providing an additional perspective on the *mems*-time relationship.
- **Validation 3: Single-Core Execution.** We further isolate CPU-level interference by pinning execution to a single core, thus minimizing variability from thread scheduling and multi-core interactions. This allows us to assess whether *mems* remains predictive under strict serial execution.

These experiments collectively examine both path-level and program-level implications of *mems*, helping assess when and where it offers meaningful insight into performance.

4.1 Experimental Setup

To evaluate the effectiveness of *mems* as a static performance metric, we conducted empirical studies across two research questions (RQ1 and RQ2), spanning multiple test environments and benchmark programs. This section summarizes the program corpus, hardware configurations, and timing methodologies used throughout the experiments.

4.1.1 Program Benchmarks

All experiments, unless otherwise stated, were performed on a suite of ten classical C benchmark programs, including sorting, searching, and array-manipulating algorithms. These programs were selected for their control-flow complexity and frequent memory access behavior. For RQ1's first validation only, we used a specially crafted illustrative program (`example.c`) designed to isolate the impact of memory operations under constant path lengths.

4.1.2 Execution Environments

The experiments were carried out on two platforms:

- **High-Performance Server (RQ1-V1, RQ1–V2, RQ2–V1)**:
  - OS: Ubuntu 20.04.6 LTS (x86_64), Kernel: 5.4.0-204-generic
  - CPU: Intel® Xeon® E5-2680 v4 CPU of 56 thread
  - Memory: 256 GB RAM
  - Timing Method: `valgrind` (`callgrind`) used to collect amplified, hardware-agnostic execution time
- **Local Machine (RQ2–V2, RQ2–V3)**:
  - OS: Windows 11 Home Edition
  - CPU: AMD Ryzen 9 7945HX, 2.50 GHz
  - Memory: 16 GB RAM
  - Compiler: MSVC with `/O0` option
  - Timing Method: `QueryPerformanceCounter` used for high-resolution wall-clock time measurement
  - RQ2–V3 additionally fixed CPU affinity to enforce single-core execution

4.1.3 Measurement Protocol

In all configurations, we systematically varied at least one input parameter: program size (denoted as `n`) to influence both path length and memory access frequency. For each configuration:

- Each program run 5 times per path, and the average execution time was recorded.
- Valgrind's `callgrind` was used to obtain stable, instruction-level time metrics.

4.2 RQ1: More Memory Accesses, Longer Execution Time

To examine the relationship between memory access counts and execution time under controlled conditions, we first study a single, well-designed C function. This function is crafted to isolate the effect of memory accesses while keeping the path length constant, making it ideal for verifying whether increased *mems* alone can influence execution time within the same program.

4.2.1 A Test Case: Memory Accesses Impact Execution Time

We begin with a simple C function designed with a loop containing a conditional branch, as shown in Listing 2. In this function, the true branch modifies an array (incurring memory accesses), while the false branch performs only arithmetic operations. This setup offers a controlled environment to analyze the impact of memory accesses on execution time, while keeping the path length consistent across different loops.

Listing 2. Function with conditional memory access

```c
void test(int n, int mode) {
  int arr[n], a = 0, b = 0;
  for(int i = 0; i < n; i++) {
    if(mode > 0) {
      arr[i] = i * 2 + arr[i];
      mode = mode - 1;
    } else {
      b = i * 3 + b;
      mode = mode - 1;
    }
  }
}
```

TABLE I
REFINED RESULTS: EXECUTION AND VALGRIND TIME ON UNINSTRUMENTED PROGRAMS

p_1	p_2	file	len	mems	t_0(ms)	vg_0(ms)
100	0	path_6	200	0	0.068	8.488
100	50	path_7	200	100	0.061	8.189
100	100	path_8	200	200	0.052	7.762
500	0	path_9	1,000	0	0.241	12.818
500	250	path_10	1,000	500	0.178	12.289
500	500	path_11	1,000	1,000	0.187	12.399
10,000	0	path_30	20,000	0	0.079	5.729
10,000	2,500	path_31	20,000	2,500	0.092	5.555
10,000	5,000	path_32	20,000	5,000	0.089	5.838
10,000	7,500	path_33	20,000	7,500	0.116	5.718
10,000	10,000	path_34	20,000	10,000	0.128	5.538
50,000	0	path_35	100,000	0	0.362	8.015
50,000	12,500	path_36	100,000	12,500	0.418	9.257
50,000	25,000	path_37	100,000	25,000	0.546	8.718
50,000	37,500	path_38	100,000	37,500	0.630	9.018
50,000	50,000	path_39	100,000	50,000	0.639	8.824
100,000	0	path_40	200,000	0	0.720	9.824
100,000	25,000	path_41	200,000	25,000	0.857	10.585
100,000	50,000	path_42	200,000	50,000	1.051	11.500
100,000	75,000	path_43	200,000	75,000	1.201	12.427
100,000	100,000	path_44	200,000	100,000	1.361	14.158
200,000	0	path_45	400,000	0	1.319	15.259
200,000	50,000	path_46	400,000	50,000	1.662	16.742
200,000	100,000	path_47	400,000	100,000	2.077	18.213
200,000	150,000	path_48	400,000	150,000	2.467	19.587
200,000	200,000	path_49	400,000	200,000	2.544	21.368

Table I presents the refined experimental results collected from uninstrumented programs. Each row corresponds to a specific execution path controlled by two input parameters:

- **p_1**(n): The first parameter, representing the input size of the program, corresponding to the variable `n` in the snippet.
- **p_2**: The second parameter, denoted as `mode` in the program, controls whether the execution path takes the memory-intensive (true) branch or the lightweight (false) branch during loop iterations.

The remaining columns report various metrics collected during execution:

- **t_0(ms)**: The actual execution time (in milliseconds) of the original, uninstrumented program.
- **vg_0(ms)**: The execution time (in milliseconds) as reported by `valgrind`'s `callgrind` tool, providing a hardware-agnostic proxy for instruction cost.

These metrics provide insight into how the number of memory operations correlates with execution time under different program scales and path conditions. For higher values of n, where memory operations become significant, a consistent proportional relationship was observed between increased *mems* and longer execution times, supporting the hypothesis that *mems* can be a significant indicator of execution time in similar path lengths.

Figures 1 and 2 illustrate the relationship between *mems* and

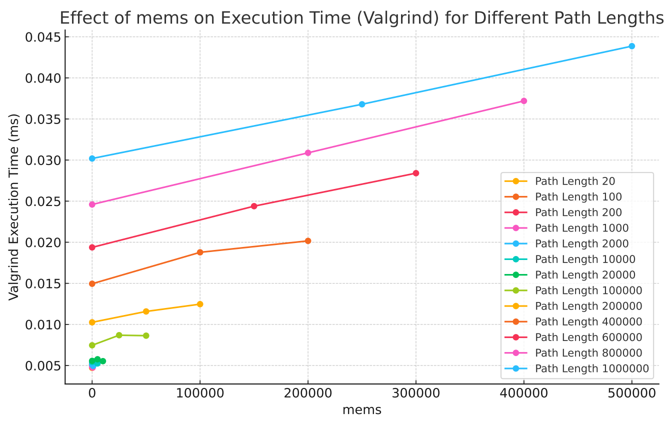

Figure 1. Graphical representation of memory access frequency and execution time correlation.

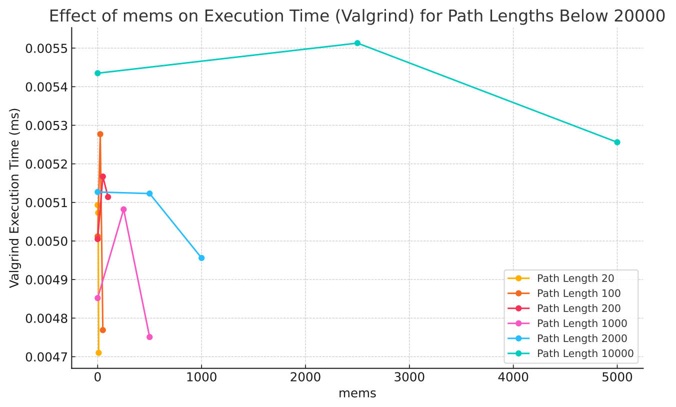

Figure 2. Analysis of path length impact on execution time across different test conditions.

TABLE II
CORRELATION COEFFICIENTS BETWEEN MEMORY USAGE AND EXECUTION TIME

Program	Correlation Coefficient	Interpretation
Array	0.99980	Very strong
Bubble	0.99980	Very strong
Insertsort	0.99996	Very strong
Sieve	0.99986	Very strong
Topo	0.99900	Very strong

Valgrind execution time under fixed path lengths. When the path length is relatively large (e.g., greater than 10,000), a clear positive correlation emerges: higher *mems* values generally correspond to longer execution times. However, for shorter paths (e.g., path length less than 1,000), this trend becomes less consistent. The execution times in these small-scale cases are more susceptible to measurement noise and system-level fluctuations, which obscure the underlying correlation and reduce the predictive utility of *mems* in such contexts.

These findings validate the proposed theory and reinforce the importance of considering memory accesses as a key factor in performance analysis, especially in scenarios with similar computational paths.

4.2.2 *mems*–Time Correlation Across Different Path Lengths

While the single test case in RQ1 demonstrated a strong correlation between memory accesses and execution time within a single program, it remains unclear whether this relationship holds across different programs with varying control structures and memory usage patterns. To investigate the generality and limitations of *mems* as a performance predictor, we conducted a comparative study involving multiple benchmark programs. This section presents the experimental setup and findings from that cross-program analysis.

The correlation coefficients were calculated to determine the strength of the relationship between memory usage and execution time across different program paths. The results are presented in Table II.

The results indicate a high degree of correlation between *mems* and execution time for the algorithms assessed within the experimental environment. This observation aligns with the results of specially constructed test cases, supporting the initial hypothesis of RQ1. The findings preliminarily validate the use of *mems* as a static indicator capable of reflecting performance costs across different paths, thereby substantiating the conjecture posed in RQ1.

Answer to RQ1 (Finding 1). These results demonstrate the feasibility of using *mems* as a performance metric to analyze the efficiency of various algorithmic paths. The strong correlations observed suggest that memory usage can reliably predict execution time, making it a valuable tool for optimizing algorithm performance in similar settings.

4.3 RQ2: *mems*–Time Correlation Across Programs

The motivating example previously presented indicates that, given identical program path lengths, the size of memory accesses (mems) impacts execution time significantly. Following Knuth's theoretical framework, it is expected that programs with similar path lengths and *mems* across different codes exhibit similar execution times. To examine whether *mems* can be a reliable performance indicator across diverse codebases, we conducted experiments using ten benchmark programs focused primarily on array-based operations such as sorting and searching algorithms.

4.3.1 Server-Side Evaluation: Partial Correlation but Inconsistent Predictability

Experiments on the server utilized the same environment as described in RQ1. Execution times measured via `valgrind` correlated strongly (correlation coefficient of 0.98) with those obtained using the Linux `time` command, validating `valgrind`'s measurements as suitable for further analysis.

As shown in Fig. 3, the combined scatter plots clearly illustrate general positive correlations between *mems*, `path length`, and execution time across different experimental conditions. Despite this general trend, substantial deviations

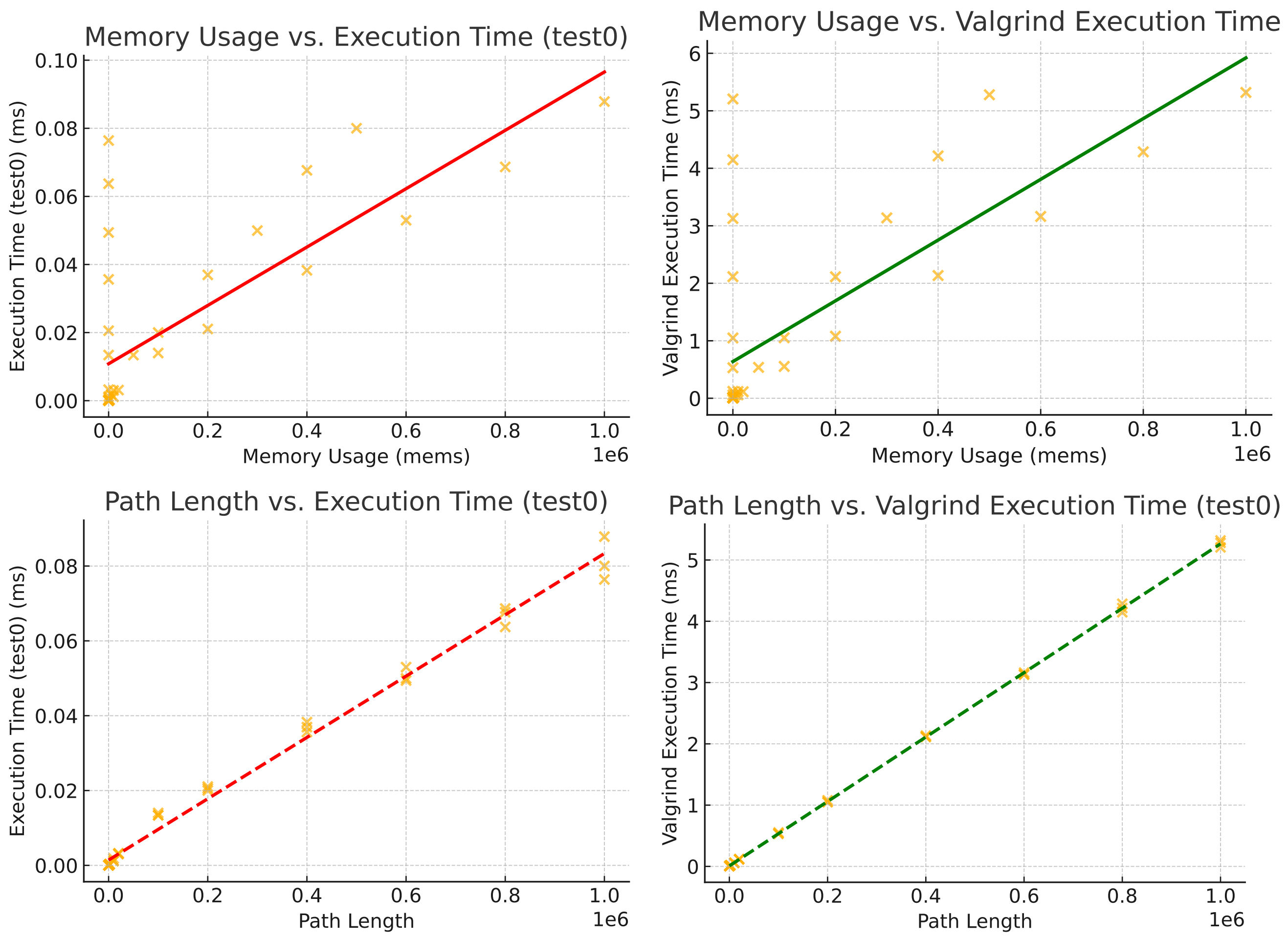

Figure 3. Scatter plots illustrating the relationship between *mems*, `path_length`, and execution time across different experimental scenarios.

and outliers remain, suggesting that neither *mems* nor `path length` individually provides a fully consistent predictor of execution time.

To further explore this issue, Fig. 4 divides the dataset into subsets grouped by similar `path length`, examining more closely the relationship between *mems* and execution time within each subgroup. Within these finer subdivisions, correlations vary significantly and sometimes deviate from the overall trend, emphasizing the complexity and variability of these relationships. These results highlight the need for more nuanced interpretation when using *mems* as a direct performance metric across diverse program paths.

Table III provides a selection of representative results from six programs tested on the server. For instance, while both `bubble.c` and `shell.c` have similar *mems* values (19800 vs. 18800), the runtime of `shell.c` is noticeably longer, implying differences in loop structure or arithmetic cost. A more dramatic deviation is seen in `change.c`, which has a long path length but very low *mems* (only 14), yet its execution time is comparable to more memory-intensive programs. This confirms that some programs, especially those with simple arithmetic or control-heavy operations but minimal memory activity, can exhibit low *mems* but still consume substantial runtime.

TABLE III
SELECTED EXPERIMENTAL RESULTS ON SERVER

Program	n	**Path Len.**	**mems**	**Time (ms)**
`bubble`	100	9,999	19,800	1.857
`change`	100	10,106	14	1.473
`shell`	1,000	12,715	18,800	2.743
`sieve`	5,000	13,175	13,089	1.267
`array`	5,000	15,002	20,000	1.479
`FFT`	2,048	18,397	118,592	7.134

Overall, these server-side experiments reveal that although *mems* is often correlated with execution time, its predictive ability is not uniform across all program types. Memory access count provides useful but incomplete insight into performance behavior. This reinforces the importance of combining *mems* with other indicators (e.g., arithmetic intensity or data dependencies) for more robust performance modeling.

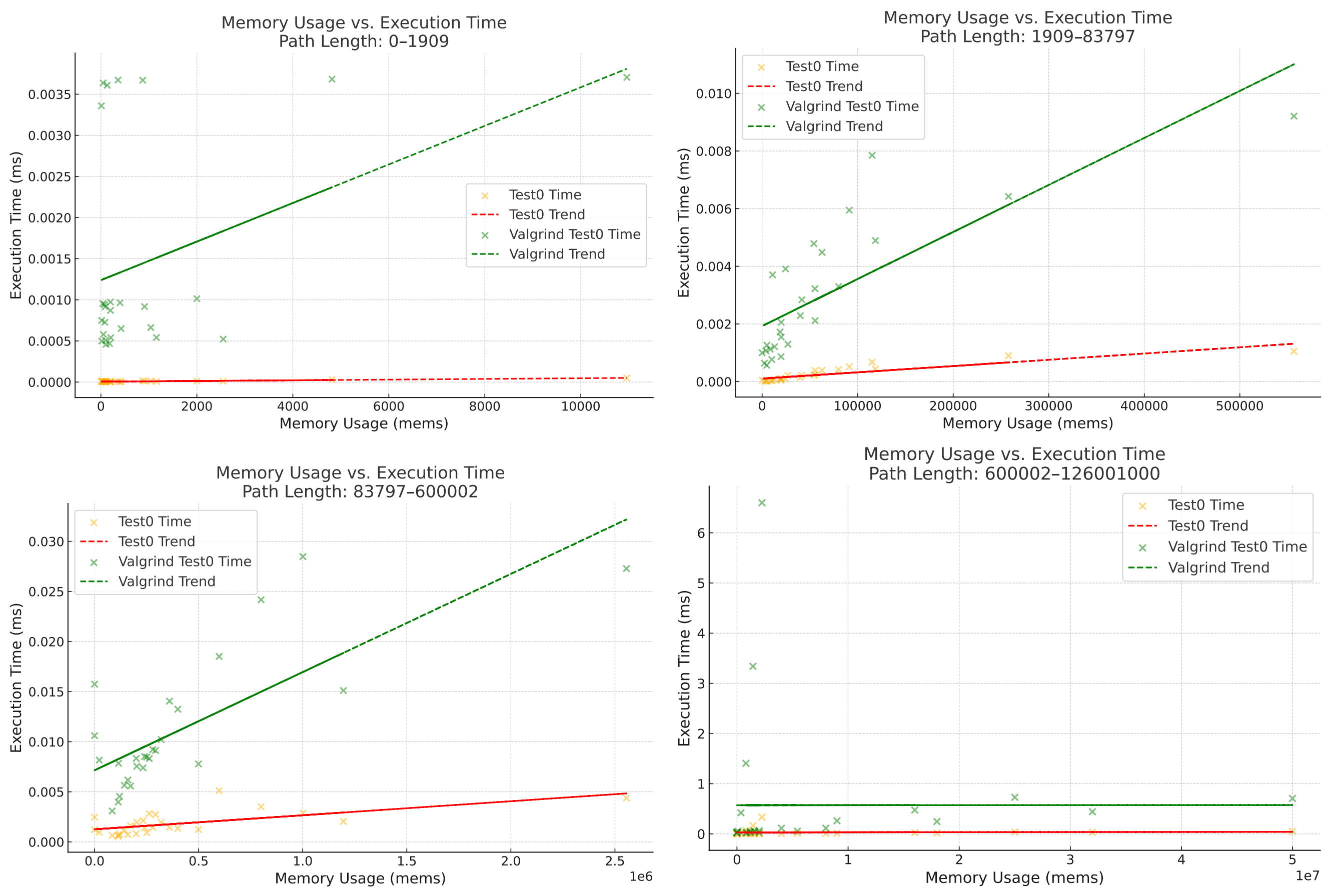

Figure 4. Scatter plots depicting the relationship between *mems* and execution time after grouping data by similar `path_length`.

TABLE IV
SELECTED EXPERIMENTAL RESULTS ON LOCAL MACHINE

Program	n	Path Len.	mems	Time (ms)
`bubble`	500	249,999	748,500	117.1
`change`	500	250,506	14	167.7
`sieve`	300,000	908,472	907,928	289.4
`bubble`	1,000	999,999	2,997,000	485.9
`insertsort`	5,000	12,507,498	25,004,998	14,913.2
`bubble`	4,000	15,999,999	47,988,000	28,550.7

4.3.2 Local Machine Experiments: Stronger Trends Under Varied Memory Loads

To account for the influence of high-performance hardware on timing variability, additional experiments were conducted on a local Windows 11 environment, disabling compiler optimizations (`/O0`) to minimize measurement noise. Execution times were measured using `QueryPerformanceCounter` instead of `valgrind`.

Table IV summarizes selected results from our local machine experiments, highlighting key findings related to the relationship between *mems* and execution time. The experiments illustrate that while paths within the same or similar programs generally demonstrate predictable trends—such as paths with significantly higher memory access counts (*mems*) typically exhibiting longer execution times—this correlation does not universally hold across different programs. For instance, comparing the programs `bubble(500)` and `change(500)`, both having nearly identical path lengths, reveals a notable anomaly: despite `bubble` featuring substantially greater *mems* (748,500 vs 14), it executes faster (117.1 ms vs 167.7 ms).

Such inconsistencies emphasize that while *mems* effectively differentiates between paths with stark contrasts in memory intensity, it does not reliably predict execution times across structurally diverse programs or different algorithmic classes. Factors such as cache locality, loop complexity, arithmetic operation intensity, and control-flow structure likely contribute to these deviations, underscoring the need for additional metrics or combined analysis strategies to achieve more accurate cross-program performance predictions.

4.3.3 Single-Core Execution: Limited Gains, Persistent Anomalies

The experimental outcomes reveal that while *mems* generally exhibit a positive correlation with execution time, this relationship is frequently inconsistent or obscured by other variables such as algorithmic characteristics and input size. Notably, input size itself demonstrated a strong correlation (0.9736) with execution time, overshadowing *mems* as a standalone predictor. Figure 5 illustrates the clear positive trend

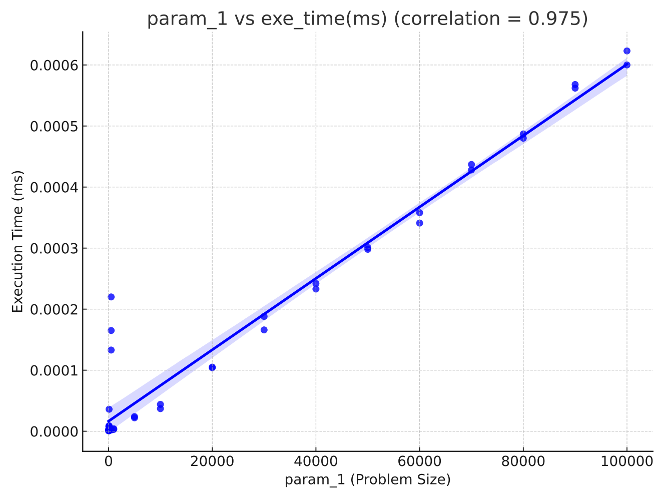

Figure 5. Scatter Plot of Input Size vs. Execution Time

TABLE V
SPEEDUP RATIOS BETWEEN SINGLE-CORE AND MULTI-CORE EXECUTIONS ACROSS DIFFERENT *mems* INTERVALS

mems	Single-core Time (ms)	Multi-core Time (ms)	Speedup Ratio
<1k	0.217	0.170	1.28×
1k–10k	2.192	1.709	1.28×
10k–100k	0.078	0.061	1.29×
100k–1M	0.599	0.550	1.09×
1M–10M	2.842	2.593	1.10×

between input size and execution time, emphasizing that larger input sizes consistently lead to longer execution times.

Considering the possibility that multi-core execution environments could introduce bias or inconsistencies into our experimental results, we restricted the experiments to single-core execution. In Windows, this single-core CPU affinity was enforced using the command:

```
cmd /c start /affinity 1
```

We reused our previously established benchmarks and compared their results in single-core versus multi-core settings.

To clearly understand how execution times vary under different memory access (*mems*) intensities, we partitioned the data into six magnitude-based intervals: $< 1K$, $1K$–$10K$, $10K$–$100K$, $100K$–$1M$, $1M$–$10M$, and $> 10M$. Figure 6 visually presents the execution time distributions for multi-core and single-core environments across these intervals.

From Fig. 6, it can be observed that multi-core execution consistently outperforms single-core execution across all intervals. Nonetheless, the performance gains differ notably among intervals. Specifically, the largest average speed-up ratio of approximately 1.29× is observed within the intermediate load interval (10^4–10^5 *mems*). In contrast, while multi-core configurations maintain advantages at high load intervals (above 10^6 *mems*), the speed-up ratios tend to stabilize and diminish slightly due to system-level limitations such as scheduling overheads and memory bandwidth bottlenecks.

Table V provides the quantified speedup ratios when comparing execution times under single-core versus multi-core environments across different *mems* intervals. As previously noted, the multi-core configuration consistently outperforms the single-core environment, with peak acceleration of approximately 1.29× observed in the interval of 10^4–10^5 *mems*. However, for larger intervals (above 10^5 *mems*), the speedup becomes less pronounced, likely due to inherent system constraints such as memory bandwidth limitations and scheduling overhead.

TABLE VI
SINGLE-CORE EXECUTION EXAMPLES

Program	Path	Len.	mems	Time (s)
`array`	path_13	240,002	320,000	0.135
`bubble`	path_3	249,999	748,500	0.251
`change`	path_2	250,506	14	0.188
`sieve`	path_14	263,282	262,985	0.201

Though restricting execution to a single-core environment indeed impacts certain benchmarks, the overall difference is relatively modest compared to amplification techniques such as Valgrind. Still, variations exist among different programs, suggesting nuanced effects rather than a uniform scaling factor. For instance, Table VI provides detailed results comparing single-core and multi-core executions for selected representative test cases:

Similar to the multi-core experiments, comparisons involving the `change.c` program exhibit a clear positive correlation between memory access counts and execution time. Nevertheless, we observe notable exceptions. Specifically, the bubble sort program, despite exhibiting a significantly greater path length and higher *mems* than the array program, paradoxically displays less than half the execution time. Such anomalies suggest that neither single-core nor multi-core execution environments alone account fully for observed deviations. Hence, additional factors, possibly including data locality, memory caching patterns, algorithmic complexity, and even compiler optimizations, must be considered to explain these discrepancies adequately.

In summary, while single-core execution control provides additional clarity regarding the influence of CPU concurrency, it confirms that substantial variations in execution times and memory access correlations cannot be explained solely by CPU affinity. This indicates a necessity for exploring further system and algorithm-specific characteristics that influence runtime performance.

Answer to RQ2 (Finding 2). When comparing across different programs, *mems* exhibits partial correlation with execution time but lacks consistency as a standalone metric. While programs with significantly higher *mems* values often demonstrate longer runtimes, numerous counterexamples reveal that factors such as algorithmic structure, loop nesting, and data access patterns can dominate performance behavior. In many cases, execution time is more strongly correlated with input size or implementation-specific details rather than *mems* alone. Additionally, variations across platforms and compiler configurations further diminish the reliability of *mems* as a

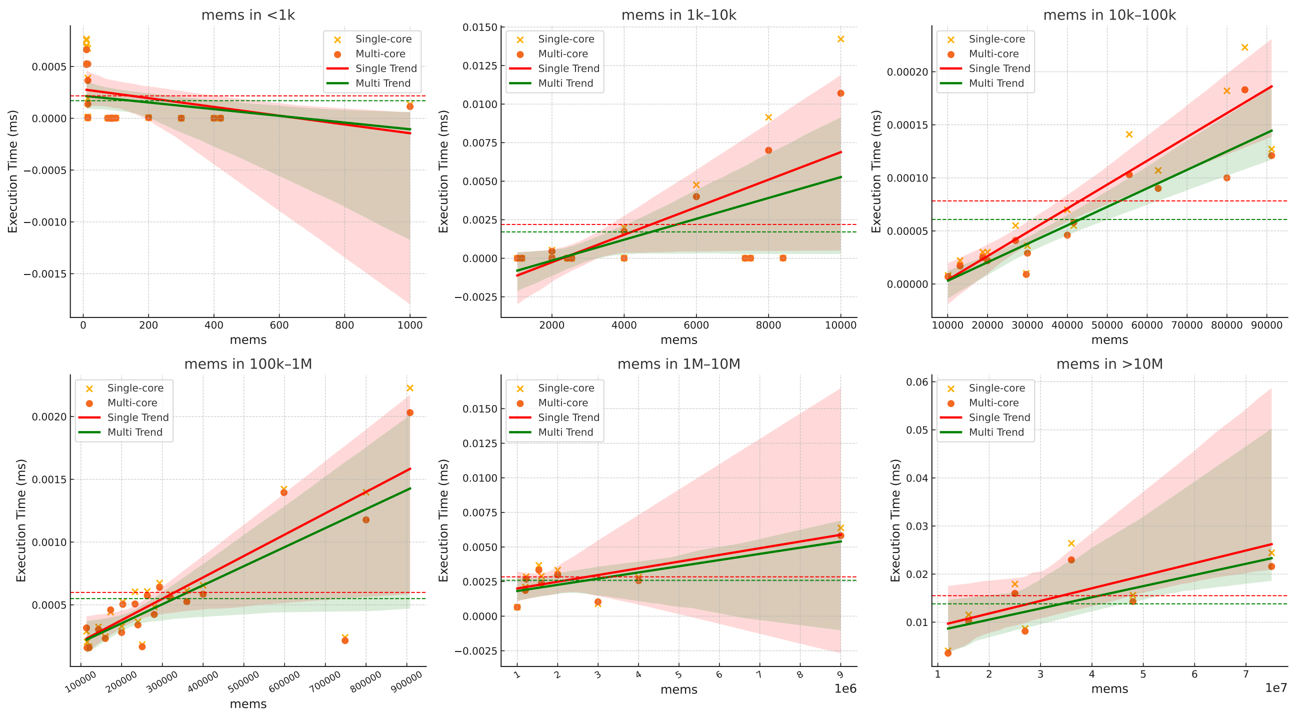

Figure 6. Execution time distributions comparing single-core and multi-core executions, grouped by memory accesses (*mems*) magnitude intervals.

cross-program performance predictor. These findings suggest that *mems* can be informative but should be interpreted in conjunction with other structural metrics such as path length or input scale to yield reliable performance insights.

4.4 Regression-Based Validation

To rigorously assess the relationship between *mems* and execution time, we apply log-log linear regression models. This approach quantifies monotonic trends and enables estimation of effect sizes and prediction confidence.

Global Analysis. Using data aggregated from all benchmarks, we fit two models:

$$\log(\text{exe\_time}) = \alpha + \beta \cdot \log(mems) + \epsilon \tag{1}$$

$$\log(\text{path\_length}) = \alpha' + \beta' \cdot \log(mems) + \epsilon' \tag{2}$$

The regression between *mems* and execution time yields $\beta = 0.439$ (95% CI : $[0.325, 0.554]$), $R^2 = 0.412$, while the result for path length gives $\beta' = 0.443$ ($R^2 = 0.338$). These moderate coefficients reflect inter-program variability and are visualized in Figure 7.

Per-Program Analysis. We also conduct per-program regression (Table VII). Most benchmarks show near-linear relationships between *mems* and execution time, with R^2 values above 0.99 and $\beta \approx 1.0$. Exceptions such as `selectsort` exhibit higher sensitivity, likely due to control-flow irregularities.

Insights. These results confirm that *mems* is a reliable static indicator for differentiating execution paths within the same program. However, its predictive power weakens across programs due to algorithmic diversity, supporting the need for complementary metrics in cross-program performance estimation.

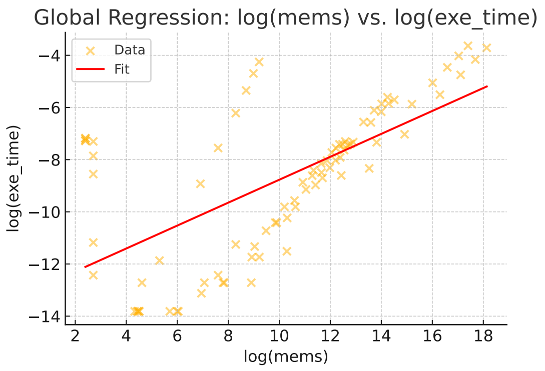

Figure 7. Global log-log regression of *mems* vs. execution time with 95% confidence interval.

TABLE VII
PER-PROGRAM LOG-LOG REGRESSION OF *mems* VS. EXECUTION TIME.

Program	Coef	95% CI Low	95% CI High	R^2	N
bubble	0.98	0.96	1.00	0.9996	8
insertsort	0.96	0.93	0.99	0.9988	9
selectsort	1.87	1.77	1.97	0.9970	8
shellsort	0.94	0.87	1.00	0.9949	8
array	0.95	0.91	0.99	0.9946	18

5. Discussion

Theoretical foundations, including Knuth's early work [6] and later analyses [1], support the use of memory access counts (*mems*) as a proxy for computational cost. From a hardware perspective, memory operations—especially those accessing main memory—incur significantly higher latency than arithmetic instructions [23]. Our RQ1 experiments reinforce this view: within the same program, paths with more memory operations consistently exhibit longer execution times.

However, RQ2 reveals the limitations of *mems* in cross-program prediction. Despite controlling for path length, discrepancies emerge due to algorithmic and structural differences. For instance, loops with better data locality or access regularity may run faster even with higher *mems*, benefiting from cache effects or vectorization. As a purely static metric, *mems* cannot capture dynamic behaviors such as cache hits, prefetching, or memory alignment.

Input size also complicates analysis. Execution time scales strongly with input parameters (e.g., n), which may overshadow the influence of *mems* in inter-program comparisons. This suggests that static memory counts are most effective for comparing paths within the same program, but less reliable across programs with differing input semantics.

Hardware factors contribute further uncertainty. While we tested across multiple platforms and controlled for compiler optimizations (e.g., `-O0`), we did not model cache hierarchy or memory latency explicitly. Notably, core count had minimal effect on correlation, implying that deeper architectural features—such as cache policy—may dominate performance variance.

Instrumentation overhead, particularly from `printf`, also introduced measurable delay, especially in short-running programs. Minimizing this overhead through buffered or low-intrusion logging would improve timing accuracy.

Lastly, our benchmark suite, though representative of classic algorithms, remains limited in scale and domain coverage. Many programs had short absolute runtimes, potentially amplifying timing noise. Future work should include larger, real-world codebases to further validate the robustness and applicability of *mems* in static performance estimation.

6. Related Work

Performance analysis has long been a central topic in software engineering. Early work such as WISE [12] and its hybrid extension by Noller et al. [13] combined symbolic execution and fuzzing to improve test coverage. However, purely static indicators often fail to reflect actual runtime behavior [15], leading to interest in hybrid approaches that trade precision for overhead and portability.

In the real-time domain, Worst-Case Execution Time (WCET) analysis [14, 18] estimates upper bounds for safety-critical code, but relies on hardware-specific modeling. Architecture-independent metrics like memory access counts (*mems*) offer a potential middle ground. Our work builds on this idea by evaluating *mems* as a lightweight, path-sensitive metric for static estimation.

Recent studies propose new strategies for worst-case analysis and symbolic reasoning. Wu and Wang [29] model worst-case cost as a posterior inference problem using sequential Monte Carlo fuzzing. Zhang and Su [30] generate executable test cases from SMT formulas, bridging formal logic and empirical evaluation.

At the system level, a utility library named LLAMA [31] offers a compile-time abstraction for memory access patterns, decoupling layout from logic. It enables precise control over memory access and layout strategies across architectures without incurring runtime overhead, which inspires memory-aware program analysis at scale.

7. Conclusion

The metric, *mems*, can be used to statically calculate a program's performance. In this paper, we provide automated solutions based on the new metric and evaluate it in real programs. We have found that in the same program, this metric has a significant positive correlation with the program execution time; while in different programs, the correlation becomes less consistent due to significant differences in memory access patterns and internal structural complexity. We hope that this metric and such findings can help the research community. In the future, we will further investigate the applicability of this metric and extend our analysis to larger and more diverse codebases to strengthen its impact on software quality assurance.

Acknowledgment

The authors would like to thank Jun Yan (ORCID: 0000-0003-2328-0781), Hao Zhang, Zhilin Li(ORCID: 0009-0008-4048-7817), Yu Xiao(ORCID: 0009-0006-5288-9616), Jiwei Yan, and the anonymous reviewers for their helpful comments and suggestions.

This work is supported by the National Natural Science Foundation of China (NSFC) under grant number 62132020.

References

[1] J. Zhang, "Performance estimation using symbolic data," in *Theories of Programming and Formal Methods*, Lecture Notes in Computer Science, vol. 8051, pp. 346–353, Springer, 2013.

[2] R. Boyer and J. Moore, "A fast string matching algorithm," *Communications of the ACM*, vol. 20, pp. 762–772, 1977.

[3] R. Cole, "Tight bounds on the complexity of the Boyer-Moore string matching algorithm," in *Proc. of the 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 224–233, 1991.

[4] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 166–176, 2012.

[5] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, 1994.

[7] S. Liu and J. Zhang, "Program analysis: From qualitative analysis to quantitative analysis," in *Proc. of the 33rd International Conference on Software Engineering (ICSE)*, pp. 956–959, 2011.

[8] F. Ma, S. Liu, and J. Zhang, "Volume computation for Boolean combination of linear arithmetic constraints," in R. A. Schmidt, Ed., *Proc. of CADE-22*, *LNCS*, vol. 5663, Springer, pp. 453–468, 2009.

[9] J. Zhang, "Quantitative analysis of symbolic execution," presented at the *28th International Computer Software and Applications Conference (COMPSAC)*, 2004.

[10] J. Zhang, "Constraint solving and symbolic execution," in B. Meyer and J. Woodcock, Eds., *Proc. of VSTTE 2005*, *LNCS*, vol. 4171, Springer, pp. 539–544, 2008.

[11] J. Zhang, S. Liu, and F. Ma, "A tool for computing the volume of the solution space of SMT (LAC) constraints," unpublished draft, Jan. 2013.

[12] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated test generation for worst-case complexity," in *Proc. of the 31st International Conference on Software Engineering (ICSE)*, Vancouver, Canada, pp. 463–473, May 2009.

[13] Y. Noller, R. Kersten, and C. S. Pasareanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *Proc. of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, The Netherlands, pp. 322–332, July 2018.

[14] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[15] H. Schnoor and W. Hasselbring, "Comparing static and dynamic weighted software coupling metrics," *Computers*, vol. 9, no. 2, p. 24, 2020.

[16] D. E. Knuth, *The Art of Computer Programming*, 2nd ed., vols. 1–3; 1st ed., vol. 4A, Addison-Wesley, Reading, Massachusetts, 1997. Fascicles of Volume 4 in progress.

[17] C. Saumya, J. Koo, M. Kulkarni, and S. Bagchi, "XSTRESSOR: Automatic generation of large-scale worst-case test inputs by inferring path conditions," in *Proc. of the 12th IEEE International Conference on Software Testing, Validation and Verification (ICST)*, pp. 1–12, IEEE, 2019.

[18] D. Bundala and S. A. Seshia, "On systematic testing for execution-time analysis," *CoRR*, vol. abs/1506.05893, 2015. [Online]. Available: http://arxiv.org/abs/1506.05893

[19] W. Chen, C. Tatsuoka, and X. Lu, "HiBGT: High-performance Bayesian group testing for COVID-19," in *Proc. of the 29th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 176–185, IEEE, 2022.

[20] F. Ma, S. Liu, and J. Zhang, "Volume computation for boolean combination of linear arithmetic constraints," in *Proc. of the 22nd International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Computer Science, vol. 5663, pp. 453–468, Springer, 2009.

[21] C. Ge and A. Biere, "Decomposition strategies to count integer solutions over linear constraints," in *Proc. of the 30th Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, pp. 1389–1395, 2021.

[22] L. Zhang, "*eppather: A Test Data Generation Tool for Unit Testing of C Programs*," GitHub Repository, 2024. Available: https://github.com/Z769018860/eppather.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[24] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA)*, pp. 452–465, ACM, 2021.

[25] Z. Xu and J. Zhang, "A test data generation tool for unit testing of C programs," in *Proceedings of the 6th International Conference on Quality Software (QSIC)*, Beijing, China, Oct. 2006, pp. 107–116.

[26] J. Zhang, "Symbolic execution of program paths involving pointer and structure variables," in *Proceedings of the 4th International Conference on Quality Software (QSIC)*, Braunschweig, Germany, Sep. 2004, pp. 87–92.

[27] D. E. Knuth, "Satisfiability and The Art of Computer Programming," in *Proc. of the 15th Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*, Trento, Italy, Jun. 2012, vol. 7317, Lecture Notes in Computer Science, p. 15, Springer.

[28] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proc. of the 38th Int'l Conf. on Software Engineering (ICSE)*, pp. 49–60, ACM, 2016.

[29] H. Wu and D. Wang, "Worst-Case Analysis is Maximum-A-Posteriori Estimation: Resource Analysis with Sequential Monte-Carlo-Based Fuzzing," *arXiv preprint arXiv:2310.09774*, 2023.

[30] C. Zhang and Z. Su, "SMT2Test: From SMT Formulas to Effective Test Cases," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 222–245, 2024.

[31] M. Kretz, M. Steuwer, F. Schlachter, H. Wehrheim, and D. Lürsen, "LLAMA: The Low-Level Abstraction for Memory Access," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 19, no. 4, pp. 1–27, 2022.