Static Analysis for Program Execution Cost Estimation

Pedro Henrique Torres Peres Garozi
Departamento de Informática, Universidade Estadual de Maringá
Maringá, Paraná, Brasil
garozipedro@gmail.com

Anderson Faustino da Silva
Departamento de Informática, Universidade Estadual de Maringá
Maringá, Paraná, Brasil
afsilva@uem.br

ABSTRACT

Program execution patterns offer valuable insights for software optimization, but accurately estimating these patterns through static analysis poses a significant challenge. While methods like machine learning and architecture simulation achieve high accuracy by utilizing extensive profiling data, they come with their own limitations in applicability and resource usage. This paper systematically evaluates control-flow analysis strategies and their combinations to estimate basic-block execution frequency. These frequency estimates are then combined with per-block cost estimations to estimate total execution cost. The investigated approach focuses on estimating branch probabilities, analyzing intra-procedural basic block frequencies, and examining inter-procedural function calls. Real execution cost is measured from a benchmark suite (cBench) to form a ground-truth profile. Correlation between cost estimates and this ground truth indicates static-method accuracy; to interpret this value, the same correlation is computed for leading simulation tools (llvm-mca, uiCA). Comparing these correlations reveals how close static analysis comes to state-of-the-art simulators. The findings show that simple heuristics achieve a correlation within 4.5% of advanced simulation tools while avoiding architecture-specific details and running in an order of magnitude less time. Additionally, analysis of how different strategies interact yields practical insights toward further improving this adaptable and efficient approach.

KEYWORDS

Static Analysis, Control-flow Analysis, Dynamic Analysis

1 Introduction

Static analysis is a cornerstone of software engineering, providing insights into program behavior without executing code [10]. Devised initially to ensure the safety and efficacy of compiler optimizations [17], it has since evolved into a versatile tool applied across various domains. One such notable application is the estimation of program execution patterns [23], a process crucial for performance optimization in scenarios where dynamic profiling is impractical or costly.

Understanding execution patterns within a program can lead to more effective optimization of distributed systems [22], high-performance computing applications [15], and highly resource-constrained environments such as embedded systems. These scenarios rely on accurate program profiles to guide decisions such as scheduling, resource allocation, or selection of optimizations. In the case of embedded systems [4] or non-standard hardware [15], where the generic compiler optimizations are often ineffective, static analysis can be instrumental in aiding optimization strategies to improve performance and reliability.

Modern static analysis techniques aim to capture programs' static profile — structural and control flow characteristics — to anticipate runtime behavior, providing insights into execution patterns that were previously accessible only through dynamic profiling. The proposal of this paper is to leverage these techniques to estimate code execution cost, allowing to circumvent the combinatorial explosion of runtime evaluations encountered in autotuning [3, 12] and optimization exploration processes [14, 16, 24], offering a cost-effective alternative to dynamic approaches.

This paper presents a modular static analysis approach for estimating program execution cost. The execution frequency analysis focus is divided into three-steps — branch probability (BP), local frequency (LF), and global frequency (GF) — as presented by Wu and Larus [23] in their classic static analysis algorithms. The approach is expanded by new heuristics for each step, and then the relationship between the strategies employed in each step is investigated — leading to an expansive field of improvements that can be made with further research. The execution frequency estimation is then combined with an individual execution cost estimation to produce the total execution cost estimation.

To assess the practicality of the static analysis approach, experiments were conducted on benchmarks from cBench. Correlation metrics compared static profile rankings against dynamic profiles, forming a structured method for identifying effective strategies. Analysis of individual strategy effects within full combinations leads to Research Question 1 investigating how the best choice for one phase (BP, LF, or GF) depends on the others. Then Research Question 2 examines the extent to which static profiles approximate dynamic profiles and how they compare to the state-of-the-art tools' profiles. The results demonstrate the approach's viability and suggest directions for improving accuracy.

This paper makes the following contributions.

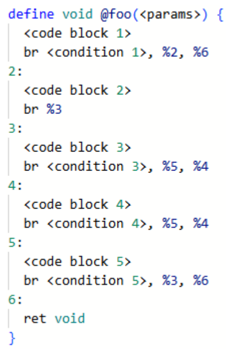
1. **Novel analysis strategies.** This work outlines a specific strategy for each analysis step. The approach not only explains how each method works but also assesses its effectiveness in estimating execution frequencies. Despite their simplicity, these strategies generally performed well, with local nesting frequency demonstrating particularly promising results.
2. **Analysis of strategy combinations.** Experiments assessed algorithms for estimating branch probability, local frequency, and global frequency. The study integrated these strategies to evaluate their strengths and weaknesses, providing valuable insights for static analysis research.
3. **Validation of execution cost estimation.** A classical static analysis approach is extended to estimate execution cost by combining execution frequency estimates with instruction cost models. Experiments show a strong correlation with

dynamic profiles, demonstrating the viability of even lightweight static cost estimation and its potential for further accuracy improvements.

The remainder of this paper is structured as follows: Section 2 explains each analysis step and the proposed heuristic strategies; Section 3 poses the research questions and delineates the methodology used to address them; Section 4 reviews related work; and finally, Section 5 concludes and outlines directions for future research.

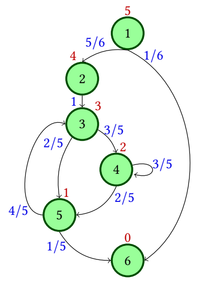
2 Background

The static analysis approach evaluated in this paper builds on the approach of Wu and Larus [23], which decomposes execution-frequency estimation into three sequential algorithms, each relying on the previous algorithm's output. For each phase — branch probability, local (intra-procedural) frequency, and global (inter-procedural) frequency — an alternative heuristic is proposed and compared against existing strategies. Section 2.1 reviews branch probability analysis and introduces the *longest* heuristic. Section 2.2 covers BB frequency analysis and presents the *localNesting* strategy. Section 2.3 describes global frequency analysis and the *globalNesting* approach. Finally, Section 2.4 explains how execution-frequency estimates are combined with instruction-cost estimation to produce the final program execution cost.



```
define void @foo(<params>) {
  <code block 1>
  br <condition 1>, %2, %6
2:
  <code block 2>
  br %3
3:
  <code block 3>
  br <condition 3>, %5, %4
4:
  <code block 4>
  br <condition 4>, %5, %4
5:
  <code block 5>
  br <condition 5>, %3, %6
6:
  ret void
}
```

(a) LLVM IR of a function (with abstracted details)



(b) BP strategy: *longest*

Figure 1: Longest path strategy application example.

2.1 Branch probability (BP)

In a CFG, most BBs terminate with a branching instruction that selects a successor BB, often based on a runtime condition (some BBs terminate with instructions such as `return` that return control to the caller). Since such conditions frequently depend on program input or execution state, branch outcomes cannot be predicted deterministically via static analysis. BP estimation assigns a probability to each outgoing edge from a BB, representing the likelihood that the branch is taken. These probabilities must sum to 1.0 for all outgoing edges of a BB.

Longest path heuristic (longest). The *longest* heuristic is provided as an alternative BP strategy to increase the available strategies for comparison and to clearly explain how a BP strategy may be implemented. To demonstrate its usage, it is applied to the CFG extracted from an LLVM IR snippet (abstracted in Figure 1a) taken from one of the evaluated benchmarks.

For a given block B in the CFG with successors $(S_0, ..., S_n)$, the heuristic estimates the BP for each edge (B, S_i) by computing the longest path $lp(S_i)$ from successor S_i to an exit block. The total path length is given by

$$total(B) = \sum_{i=0}^{n} lp(S_i) + 1$$

, and the branch probability for each successor is defined as

$$BP(B, S_i) = \frac{lp(S_i) + 1}{total(B)}$$

Figure 1b illustrates this process, where the computed $lp(B_i)$ is displayed in red and the BP(B, Si) in blue.

2.2 Local frequency (LF)

When a function is called, execution begins at its entry block, ensuring that this block runs at least once (i.e., $LF = 1$). As the entry block branches out to its successor blocks, the LF analysis distributes the LF among them based on the BP from the previous step. If a branch is a back edge, the algorithm adjusts the frequency to account for additional iterations in loops.

Local nesting heuristic (localNesting). A straightforward heuristic called *localNesting* is introduced as a strategy for analyzing LF by determining the nesting level of each BB in the CFG. The method utilizes a depth-first search (DFS) to construct a depth-first spanning tree (DFST), where each node corresponds to a BB. When a back edge is detected, the associated node is marked as a loop head, and all nodes in the DFS stack leading back to the loop head are designated as part of its loop group. The DFST is then traversed to assign a nesting level to each node based on its group affiliation. This nesting level is subsequently used to calculate the LF.

Figure 2 illustrates the DFST resulting from applying the aforementioned algorithm to the CFG depicted in the same figure. The yellow nodes highlight the loop heads. When a loop head is encountered, the nesting count (number in blue) is incremented for all nodes in the corresponding children's group (numbers in red: group/child-group). For instance, nodes 4 and 5 belong to group 1, which is associated with the children of node 3; therefore, their nesting level is initially one higher than that of node 3. After assigning the nesting level of every child node, loop heads have their nesting level increased by one. Finally, the LF of each node is computed by propagating the BP values to each node and multiplying these by their computed nesting level.

2.3 Global frequency (GF)

LF analysis estimates BB frequencies under the assumption that each function is invoked exactly once. In reality, functions in the CG may be invoked multiple times — or not at all. The last step of the execution frequency analysis is the inter-procedural GF analysis that builds upon LF estimation by capturing function call behavior across procedure boundaries; this allows the LF of blocks within these functions to be adjusted based on their invocation counts.

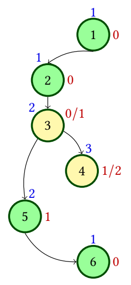


Figure 2: LF strategy: *localNesting*

Specifically, the local call frequency is calculated as the sum of the LF of all BBs in function f that contain calls to function g. Then the employed GF strategy uses the local call frequency as initial weights in the CG edges to determine each function's invocation frequency, that is, how many times they are estimated to be called in a typical execution of the program. Finally, the LF is multiplied by the invocation frequency, yielding the GF of each BB.

Global nesting heuristic (globalNesting). The *globalNesting* strategy is an inter-procedural version of *localNesting*, this time operating on the CG by considering the nodes to be functions and the edges to be calls between functions. Figure 3 illustrates the CG for a recursive function invoked by the main function. Due to the recursion, this function is treated as a loop head, resulting in an increase of its nesting level. Lastly, the local call frequency is propagated from the main function through the edges in the CG (similarly to how BP is propagated by LF) and the resulting invocation frequency of each function is multiplied by its nesting level.

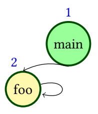


Figure 3: GF strategy: *globalNesting*

2.4 Execution cost estimation

After the third step, the GF of each block in the code is determined. This already valuable information allows focusing optimization and debugging efforts in the most executed portions of the code. For the purpose of this work, there is still a piece missing: a single execution cost estimation. By estimating a single execution cost for each block and estimating its execution frequency, it is now possible to multiply both values to determine an estimate of the BB's total execution cost. By adding together the total execution cost of every BB in the code, it is possible to estimate the total execution cost for the whole code.

The cost estimations provided by LLVM's TargetIRAnalysis (*latency*, *recipthroughput*, and *codesize*) were evaluated, and *codesize* yielded the best results; therefore, it was used as the instruction cost estimation. Each basic block's single execution cost was then computed by summing the *codesize* of its instructions.

3 Evaluation and analysis

This section measures the viability of the static analysis approach presented in Section 2 for execution cost estimation. Section 3.1 states the research questions. Section 3.2 describes the evaluation methodology. Section 3.3 details the experimental setup in terms of hardware and software. Section 3.4 explains the metrics used to measure the correlation between static and dynamic profiles. Section 3.5 defines each strategy employed. Finally, Section 3.6 presents results for RQ1, and Section 3.7 presents results for RQ2.

3.1 Research questions

Research question 1 (RQ1). Is the best strategy for each analysis step (BP, LF, GF) independent of the strategies selected for the other steps in the combination?

This question aims to facilitate experiments in evaluating the relative quality of a given strategy by understanding how the impact of a strategy in a combination affects its overall quality.

Research question 2 (RQ2). To what extent do static profiles produced by the three-step static analysis investigated in this work correlate with dynamic profiles measured from actual code execution?

This question evaluates 21 analysis strategy combinations alongside the heavier simulation-based tools against a benchmark suite to analyze the yielded correlation.

3.2 Methodology

This section offers a high-level view of the employed methodology. As illustrated in Figure 4, the employed methodology can be broken into:

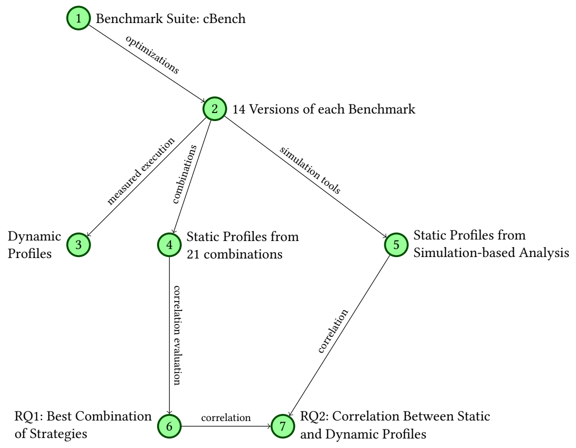

1 Benchmark Suite: cBench
optimizations
2 14 Versions of each Benchmark
measured execution
combinations
simulation tools
Dynamic Profiles 3
4 Static Profiles from 21 combinations
5 Static Profiles from Simulation-based Analysis
correlation evaluation
correlation
RQ1: Best Combination of Strategies 6
correlation
7 RQ2: Correlation Between Static and Dynamic Profiles

Figure 4: Methodology diagram

(1) **Benchmark suite.** To evaluate the accuracy of execution cost estimations, 18 benchmarks from cBench [1] were selected; others were excluded due to analysis failures with the evaluated tools.

[1]https://sourceforge.net/projects/cbenchmark/files/cBench/V1.1/

(2) **Benchmark versions.** Each program was compiled with 14 different optimization sequences — O0, O1, O2, O3, and 10 random sequences — resulting in a total of 252 (14×18) distinct compiled versions. The compilation was performed step-by-step, generating the files in the format required by each tool: IR, assembly, and object code. These 14 variations of each benchmark are referred to as *code sets*.
(3) **Dynamic profiles.** Each of these versions was executed with representative input, and perf [2] was utilized to collect CPU cycle usage — the average of three runs was used; this is the default setting in many benchmark suites. This provided the ground truth, allowing a comparison of the static profiles estimated by each tool against the dynamic profiles obtained with perf, ensuring a fair and meaningful evaluation.
(4) **Static profiles from strategy combinations.** The strategies employed in each step of the analysis are applied to every benchmark version, generating their static profiles — in this case, an estimation of their total execution time. A listing of all evaluated strategies by each analysis step is given in Section 3.5.
(5) **Static profiles from state-of-the-art tools.** To better understand the quality of the static profiles obtained by each combination, state-of-the-art tools — llvm-mca (Machine Code Analyzer) and uiCA (uops.info Code Analyzer) — are employed to estimate the static profile using the heavier simulation-based approach. These tools simulate the target architecture executing the code and predict its throughput.
(6) **Analyzing strategy effects.** Six targeted experiments are performed — two for each heuristic phase (BP, LF, GF) — to isolate and measure the impact of swapping a single strategy within an otherwise fixed combination. This paves the way for the answer to RQ1 in Section 3.6.
(7) **Static profile correlation evaluation.** Multiple correlation metrics are used to compare the ranking of each combination's static profile against the true dynamic profile and against simulation-based predictions. This evaluation is expanded in Section 3.7 and quantifies the combinations' accuracy, positions them relative to state-of-the-art simulation tools, and demonstrates the validity of the approach.

3.3 Hardware & software

All experiments were conducted on an Intel i7-3770, which features 4 cores and 8 threads, running at a clock speed of 3.40 GHz. The system was equipped with 16 GB of RAM and operated on Ubuntu 22.04.4 LTS. Benchmark programs were compiled using Clang version 15.0.7 with the -O0 optimization level, followed by additional optimization sequences applied through llvm-opt. The execution cost was measured using perf version 6.8.12, which collected user CPU cycle counts from three separate runs and averaged the results. Additionally, llvm-mca version 15.0.7 and uiCA were executed on the assembly output of the optimized code, both configured to simulate the Ivy Bridge architecture in order to align with the measurements obtained from perf.

[2]https://man7.org/linux/man-pages/man1/perf.1.html

3.4 Metrics

In this context, precision focuses on maintaining the relative order of execution times rather than achieving exact cycle counts. The success of the estimation method relies on accurately identifying which programs are faster or slower, ensuring that the rankings of CPU cost observed through tools like perf are preserved, even if the absolute values differ. Thus, maintaining the order of costs is more crucial than achieving absolute cycle precision. Kendall's tau (τ) coefficient [1] is used to quantify this precision by measuring the agreement between the estimated and actual orderings, ensuring that we can rely on the relative performance differences. This metric is commonly used in the literature [2] to evaluate ranking consistency, highlighting its suitability as a basis for comparisons. To provide a basis for comparison, the more well-known Pearson and Spearman correlations are also provided [7].

3.5 Combinations of strategies

The static analysis approach proposed in this work is to turn the algorithms employed in each analysis step into independent strategies that may be easily swapped for experimentation. The selection of a BP strategy, LF strategy, and GF strategy is called a combination and is denoted as <BP> + <LF> + <GF>. For instance, the strategies described in Section 2 would be the combination *longest* + *localNesting* + *globalNesting*. Since Wu & Larus [23] and LLVM refer to their analyses by generic names, here we give them unique names to be more distinguishable in this context. Table 1 presents the evaluated strategies, and each of them is described in the following paragraphs.

Table 1: Summary table of evaluated strategies

BP	LF	GF
hierarchical	*massPropagation*	*one*
fusion	*localPropagation*	*globalPropagation*
longest	*localNesting*	*globalNesting*
equal		

Evaluated BP strategies. In this paper 4 BP strategies were evaluated: *hierarchical*, *fusion*, *equal*, and *longest*. The *hierarchical* strategy is implemented in LLVM. It first attempts to utilize dynamic profiling data; if this data is unavailable, it resorts to known block properties. If neither of these options is successful, it sequentially applies heuristics until a match is found. The *fusion* strategy implements Wu and Larus's Algorithm 1 [23], estimating BP by combining multiple heuristics derived from experimental data that may apply to the same branch. Additionally, we consider a baseline approach called *equal*, which assigns a uniform probability to each branch. For example, in a node with four successors, each branch would have a probability of 25%. Finally, we examine the *longest* strategy, which serves as a simple, illustrative example of BP estimation, as detailed in Section 2.1.

Evaluated LF strategies. The 3 evaluated LF estimation algorithms are *massPropagation*, *localPropagation*, and *localNesting*. The *massPropagation* algorithm, which is a pass implemented in LLVM, computes LF using a bottom-up approach with directed acyclic graphs (DAGs) for each loop. Each DAG is assigned a mass, which is then utilized to calculate the loop scales. The *localPropagation* strategy is an implementation of Wu and Larus' Algorithm 2 [23]; it determines LF by propagating BP within the CFG and adjusts cyclic probabilities when it detects back edges. Finally, *localNesting* is a straightforward analysis that adjusts the BP based on the nesting level of loops, as presented in Section 2.2.

Evaluated GF strategies. Due to the lack of an inter-procedural analysis pass in LLVM, only 2 methods for GF estimation are evaluated in this paper: *globalPropagation* and *globalNesting*. The *globalPropagation* strategy is an implementation of Wu and Larus's Algorithm 3 [23], which is an inter-procedural adaptation of their Algorithm 2 for LF. Similarly, *globalNesting* follows the same approach as detailed in Section 2.3. We also examine the effects of completely omitting GF estimation, a configuration we refer to as *one*.

3.6 Strategy influence analysis (RQ1)

The objective of this section is to answer Research Question 1 by determining if a single, superior strategy can be identified for each analysis phase — BP, LF, and GF estimations — regardless of the strategies used in other phases.

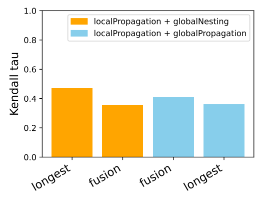
To better state the problem, consider two strategies, A1 and B1, for different analysis steps; their partial combination, A1 + B1, lacks one strategy. Let C and D be two candidates for the remaining step, forming full combinations A1 + B1 + C and A1 + B1 + D. If evaluation shows C outperforms D in the context of A1 + B1, but under a different partial combination A2 + B2 (with $A2 \neq A1 \lor B2 \neq B1$) D outperforms C, then neither C nor D can be deemed universally better: the effectiveness of a strategy depends on the other strategies in the combination.

Figure 5 illustrates the experiment. The subsequent paragraphs show for each analysis step that evaluating a strategy's effectiveness requires evaluating its interaction with the other strategies in the combination.

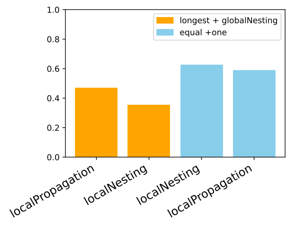
BP strategy. In the partial combination *localPropagation* + *globalNesting* (represented by the orange bars in Figure 5a), the *longest* BP strategy shows a higher correlation. On the other hand, when considering *localPropagation* + *globalPropagation* (shown by the blue bars in Figure 5a), the *fusion* strategy outperforms the *longest*. This suggests that the best BP strategy depends on the specific LF and GF methods used.

LF strategy. When using the partial combination *longest* + *globalNesting* (represented by the orange bars in Figure 5b), the most effective LF for completing the partial combination is *localPropagation*. In contrast, when employing the partial combination *equal* + *one* (shown by the blue bars in Figure 5b), *localNesting* performs better than *localPropagation*. These results indicate that the optimal LF strategy depends on the specific combinations of the BP and GF configurations.

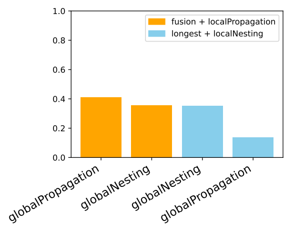
GF strategy. When using *fusion* + *localPropagation* (as shown in Figure 5c with the orange bars), the *globalPropagation* strategy demonstrates a higher correlation. Conversely, when employing



(a) The best BP changes.



(b) The best LF changes.



(c) The best GF changes.

Figure 5: Verification of best strategy in each analysis step.

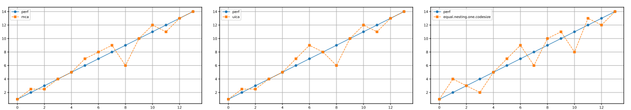
longest + *localNesting* (indicated by the blue bars in Figure 5c), the *globalNesting* strategy shows improved accuracy. Therefore, the optimal GF strategy depends on the chosen BP and LF methods.

The experiments show that the effectiveness of each strategy relies on the full combination across phases, indicating the importance of assessing the synergy between strategies. While this requires exploring different combinations to find the most effective configuration for a given context, it also shows that a strategy that performs modestly on its own can achieve much better results when combined with more compatible techniques. These findings open the door to developing synergistic approaches that leverage the interactions between branch probability, local frequency, and global frequency estimators. Ultimately, this demonstrates that the best strategy is dependent on the context.

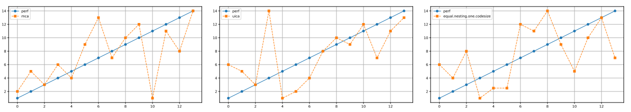
3.7 Cost-estimation analysis (RQ2)

The goal of this section is to evaluate the correlation between the static profile obtained by the strategy combinations and the actual dynamic profile. To assess the quality of the analysis, the execution cost estimates are compared with those produced by state-of-the-art simulation-based tools, thereby providing insights into their reliability, precision, and overall performance for execution cost estimation.

Reliability. To evaluate the reliability of our results, Figure 6 presents, for llvm-mca, uiCA, and the top combination, the benchmarks where each achieves its highest (figures 6a, 6b and 6c) and lowest (figures 6d, 6e and 6f) correlation with the dynamic profile. All three tools achieve very high correlations in favorable scenarios, demonstrating their potential to replace expensive dynamic analysis in certain contexts. Meanwhile, the worst correlations remain moderate, ensuring a base level of reliability even in less favorable situations.



(a) mca $\tau = 0.906$ **(b) uiCA** $\tau = 0.884$ **(c) top** $\tau = 0.802$



(d) mca $\tau = 0.472$ **(e) uiCA** $\tau = 0.406$ **(f) top** $\tau = 0.309$

Figure 6: Comparison of best and worst correlations for each analysis. The y-axis shows normalized execution time for each code in the *code set* (x-axis). The blue line represents perf's measurments, while the orange lines show the cost estimates.

Correlation with dynamic profile. The correlation between the dynamic profile obtained with perf and the static profile obtained with the top-scoring combination (*equal* + *localNesting* + *one*), the state-of-the-art simulation-based tools (llvm-mca and uiCA), and the baseline execution cost estimation without execution frequency estimation (one-pure) is shown in Figure 7. As anticipated, llvm-mca and uiCA yield the highest correlations; however, the leading combination is highly competitive, trailing only 4.43% beyond llvm-mca and 3.1% beyond uiCA. Notably, the range between the worst and best correlations for llvm-mca, uiCA, and the top combination (*equal* + *localNesting* + *one*) is quite similar. The one-pure experiment, which bypasses execution frequency estimation and is solely based on instruction cost estimation (*codesize* in this case), shows a high correlation. This is not surprising, as code size often serves as a reliable indicator of execution cost and is just 4.79% below the top combination.

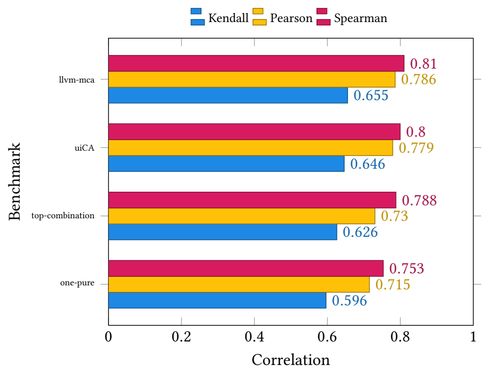


Figure 7: Comparison of the correlation (using three metrics) between the top scoring combination, the state-of-the-art tools (llvm-mca and uiCA) and the baseline (one-pure).

Performance of the analyses. Although the simulation-based approach is more precise on average, the execution of the experiments revealed it to be one order of magnitude slower. Figure 8 shows the total execution time for the *code set* of each benchmark. These findings indicate that even simple static analysis strategies can match the performance of complex simulation-based tools while being far more efficient. For example, in cBench's office_rsynth *code set*, the best static configuration averages 0.057s with a Kendall's correlation of 0.516, compared to uiCA's 14.644s (correlation 0.428) and llvm-mca's 4.197s (correlation 0.472). These results demonstrate that static analysis not only provides a scalable alternative but can even surpass simulation-based tools in accuracy for some *code sets*, making it a promising direction for further research. Note that uiCA's timings are not included in Figure 8 because its execution for some benchmarks required more memory than was available in the experimental setup; this led to high execution times due to use of the swap memory, but the executions that could be properly measured had timings similar to llvm-mca.

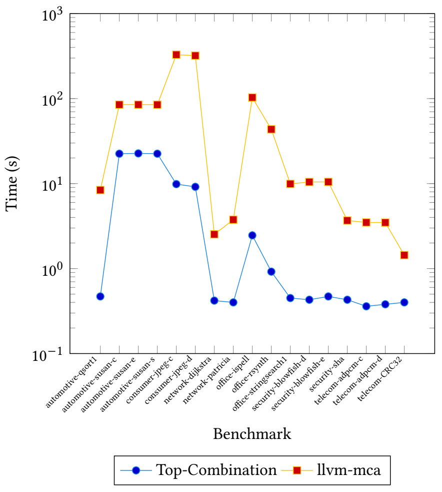


Figure 8: Timings with a logarithmic scale.

Pitfalls of control-flow static analysis. Despite being blazing fast and reaching a high correlation, the fact that most strategies performed worse than their respective baseline is disappointing. The results presented in Figure 9 indicate that configurations that do not include a separate GF analysis phase produced more accurate estimations in our experiments. This improvement is likely due to the fact that the GF phase can propagate and amplify existing errors from the LF estimation. Also, even in configurations with strong correlations, the BP heuristics performed slightly worse than the baseline *equal* strategy, which applies no heuristics. This suggests that the BP strategies evaluated may struggle to model branch behavior accurately without profiling data, highlighting the need for more refined heuristics specifically designed for modern programming practices or the integration of evidence-based branch prediction techniques.

The *localNesting* strategy consistently ranks among the top three configurations, indicating that properly modeling loop behavior is a key to improving strategies' precision. In contrast, the *localPropagation* strategy performs worse than the *one-pure* baseline, suggesting that traditional algorithms [23] may need to be updated to better align with contemporary programming practices. Additionally, LLVM heuristics — *hierarchical* and *massPropagation* — perform poorly in this static context as they were designed to utilize dynamic profiling data; also, because of implementation details, they could not be evaluated in combination with other strategies.

These findings highlight the necessity for new, well-designed analysis strategies that encourage a cooperative interplay between heuristics, leveraging their strengths while minimizing individual weaknesses. The key aspect in achieving a high correlation may rely on the performance of the evaluated strategies. Since their execution cost is so low, there is plenty of room for deeper and more accurate analyses.

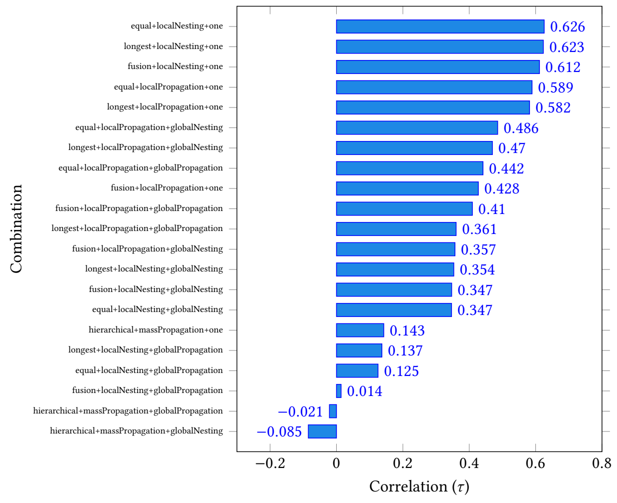


Figure 9: Average Kendall's Tau (τ) correlation for each combination evaluated.

Pitfalls of the simulation-based approach. Although uiCA [2] reports an impressive Kendall's tau correlation higher than 0.9 and llvm-mca achieves over 0.8 for the BHive benchmark, the experiments on cBench indicate that simulation-based tools heavily depend on assumptions that may not hold in unfiltered benchmarks. Despite the resulting reduction in correlation, these tools primarily target throughput prediction, and their ability to maintain a high correlation remains a notable success.

4 Related work

Worst case analysis. Some static analysis techniques model program behavior by extracting execution constraints and solving cost equations to estimate resource consumption in the worst case, which are related to the worst-case execution time (WCET) and worst-case energy consumption (WCEC) problems. Previous research, such as energy requirements compliance, demonstrates this approach [20]. While these techniques are effective for bounding worst-case resource usage and estimating asymptotic complexity, they have limitations for average-case estimations. Although high complexity can make these programs impractical for large inputs, algorithms with better constant factors may outperform those with lower theoretical complexity in real-world scenarios, as worst-case scenarios might be infrequent.

Simulation-based analysis. Simulation-based analysis [2, 9] estimates performance by simulating the execution of machine code using detailed hardware models and scheduling information. Tools such as llvm-mca [3], OSACA [9], and uiCA [2] parse assembly code into machine instructions, simulate their execution, and utilize target-specific scheduling models to calculate metrics like instructions per cycle (IPC), throughput, and resource pressure. This process helps identify performance bottlenecks and provides insights into the finer aspects of processor behavior. Although this method is highly accurate, it is computationally demanding for large programs, requiring significant CPU time and memory.

Machine learning. Machine learning (ML) has become increasingly popular for execution cost estimation, providing a data-driven alternative to static analysis. Techniques like Ithemal [11] and GRANITE [19] learn to predict performance directly from execution data by utilizing methods such as deep neural networks and graph neural networks. This approach eliminates the need for explicit execution or manual cost modeling. In contrast, CLUTCH [18] employs a different strategy by using an ensemble of ML techniques to estimate execution time for scientific simulations. While ML models can achieve high accuracy, they do shift the computational burden to the training phase. Additionally, the effectiveness of pre-trained models largely depends on their ability to generalize across different domains. Poor generalization can result in higher computational demands if users are required to train models specific to their domain.

5 Conclusion

This paper presents a modular static analysis approach based on control flow that divides execution frequency estimation into three analysis steps: BP, LF, and GF. Allowing each phase to be implemented using different strategies enables flexible experimentation with various algorithmic combinations tailored to different code characteristics. After the execution frequency of each block is estimated, it is then combined with the estimation for a single execution of the block. The evaluation shows that even simple strategies can achieve results close to those produced by complex simulation-based tools, with differences of approximately 4.5% while being on average 10x faster than simulation-based tools.

Findings indicate that overly simplistic heuristics may reduce estimation accuracy. Nonetheless, their competitive performance on benchmarks suggests that more advanced heuristic designs could further enhance precision. This static analysis approach requires far less compute time than cycle-accurate simulation or ML-model training and provides greater flexibility than worst-case execution time (WCET/WCEC) analyses, which often overestimate cost and may not reflect typical behavior in practice.

The strong correlation observed between the static profiles generated by the best combinations and the actual dynamic profiles confirms the primary goal of this work — this approach is an efficient and reliable replacement for dynamic profiling in resource-constrained environments. Each strategy can be adapted for various

[3]https://llvm.org/docs/CommandGuide/llvm-mca.html

applications, such as guiding compiler optimizations by estimating the impact of code transformations.

Future work. (i) This work laid fertile ground for further studying both new execution frequency estimation strategies and the synergy between their combinations. The *localNesting* strategy showed a promising path by identifying the nesting level of loops. Further research may combine other static analysis techniques like constant propagation [21] and value-range propagation [13] or even ML approaches like evidence-based BP [5] and decision trees [6] to further refine the accuracy of estimations. (ii) Also, only the default LLVM estimations for instruction execution cost were evaluated. The correlation between the *codesize* estimation and the dynamic profiles even without execution frequency estimations shows that this is one of the main aspects in reaching a high correlation. New heuristics or ML techniques may be applied to reach even higher precision. (iii) Besides, while this work focuses on execution time estimation, by modifying the instruction cost estimation goal, the approach can be readily extended for other purposes, such as energy consumption estimation [8].

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] Hervé Abdi. 2007. The Kendall rank correlation coefficient. *Encyclopedia of measurement and statistics* 2 (2007), 508–510.

[2] Andreas Abel and Jan Reineke. 2022. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[4] Jose Antonio Ayala-Barbosa and Paul Erick Mendez-Monroy. 2022. A new preemptive task scheduling framework for heterogeneous embedded systems. In *Proceedings of the 2022 8th International Conference on Computer Technology Applications*. 77–84.

[5] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 188–222. doi:10.1145/239912.239923

[6] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. 2005. Using decision trees to improve program-based and profile-based static branch prediction. In *Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005, Singapore, October 24-26, 2005. Proceedings 10*. Springer, 336–352.

[7] F Essam, Hashash El, and Shiekh Raga Hassan Ali. 2022. A comparison of the pearson, spearman rank and kendall tau correlation coefficients using quantitative variables. *Asian J. Probab. Stat* (2022), 36–48.

[8] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems* (Sankt Goar, Germany) *(SCOPES '15)*. Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/2764967.2764974

[9] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. 2018. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*. IEEE, 121–131.

[10] Haoxuan Li, Paul De Meulenaere, Ken Vanherpen, Siegfried Mercelis, and Peter Hellinckx. 2020. A hybrid timing analysis method based on the isolation of software code block. *Internet of Things* 11 (2020), 100230.

[11] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[12] Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. 2025. Towards Efficient Compiler Auto-tuning: Leveraging Synergistic Search Spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 614–627.

[13] Jason R. C. Patterson. 1995. Accurate static branch prediction by value range propagation. *SIGPLAN Not.* 30, 6 (June 1995), 67–78. doi:10.1145/223428.207117

[14] Lana Scravaglieri, Ani Anciaux-Sedrakian, Olivier Aumage, Thomas Guignon, and Mihail Popov. 2025. Compiler, Runtime, and Hardware Parameters Design Space Exploration. In *IPDPS 2025-39th IEEE International Parallel and Distributed Processing Symposium*.

[15] Lana Scravaglieri, Mihail Popov, Laércio Lima Pilla, Amina Guermouche, Olivier Aumage, and Emmanuelle Saillard. 2023. Optimizing performance and energy across problem sizes through a search space exploration and machine learning. *J. Parallel and Distrib. Comput.* 180 (2023), 104720.

[16] Anderson Faustino da Silva, Bernardo NB De Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 47–58.

[17] Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević, and Sonja Ristić. 2020. Static code analysis tools: A systematic literature review. In *Ann. DAAAM Proc. Int. DAAAM Symp*, Vol. 31. 565–573.

[18] Young-Kyoon Suh, Seounghyeon Kim, and Jeeyoung Kim. 2020. Clutch: A clustering-driven runtime estimation scheme for scientific simulations. *IEEE Access* 8 (2020), 220710–220722.

[19] Ondřej Sýkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh. 2022. Granite: A graph neural network model for basic block throughput estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 14–26.

[20] Kai Vogelgesang, Phillip Raffeck, Peter Wägemann, Thorsten Herfet, and Wolfgang Schröder-Preikschat. 2024. WIP: Towards a Transactional Network Stack for Power-Failure Resilience. In *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*. IEEE, 803–806.

[21] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. doi:10.1145/103135.103136

[22] Carl Witt, Marc Bux, Wladislaw Gusew, and Ulf Leser. 2019. Predictive performance modeling for distributed batch processing using black box monitoring and machine learning. *Information Systems* 82 (2019), 33–52.

[23] Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose, California, USA). Association for Computing Machinery, New York, NY, USA, 1–11.

[24] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. 2020. YACOS: a complete infrastructure to the design and exploration of code optimization sequences. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*. 56–63.