AC Loop: An LLM-Based End-to-End Auto Calibration Framework

Hao Zhen Qingxuan Kang* **Yungang Bao Trevor E. Carlson*†**
State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, China
*National University of Singapore, Singapore †Google
{zhenhao}@ict.ac.cn,{k.qingxuan}@u.nus.edu,{baoyg}@ict.ac.cn,{tcarlson}@comp.nus.edu.sg

***Abstract*—High-level architectural simulators are widely used for performance exploration during hardware design. However, performance differences from the RTL design can lead to inaccurate estimates. Existing approaches address this through model calibration to ensure faithful reflection of the target hardware. One recent calibration methodology proposes single-feature-attribution microbenchmarks as a way to target microarchitecture structures. But, these microbenchmarks are manually designed requiring substantial engineering effort and remaining prone to errors.**

To improve calibration efficiency, we introduce *AC Loop*, an LLM-driven end-to-end automated calibration framework. The AC loop leverages a modern LLM to generate calibration-oriented microbenchmarks. Given sufficient background information and structured guidance, the LLM can effectively produce microbenchmarks that enable rapid calibration. AC loop provides a fully automated, end-to-end pipeline that integrates calibration, microbenchmark execution, and result analysis end to end.

We evaluate AC Loop on the RTL and gem5 modules of the OpenXiangShan platform. The results show that it effectively generates microbenchmarks, corrects misaligned parameters, and detects fine-grained discrepancies. In one case study, the AC Loop took just 7.7 minutes to automatically generate appropriate microbenchmarks for characterizing Load Queue capacity, consuming approximately ten thousand input and output tokens.

I. Introduction

Architectural simulators are essential tools for performance analysis and design-space exploration during hardware development. They enable early evaluation of new microarchitectural features before such a feature is implemented in RTL. However, without proper calibration, a simulator can compromise the validity of such evaluations [6]. This issue arises when the simulator fails to faithfully capture critical behaviors and interactions that occur in the RTL implementation. Typical sources of modeling inaccuracies include outdated parameter configurations, idealized or incomplete modeling of resource contention, and incorrect assumptions regarding the relative importance of some microarchitectural features.

Model calibration ensures that the simulator consistently reflects the intended microarchitectural responses. To enable a tractable comparison between the simulator and the RTL implementation, prior work primarily relies on microbenchmarks [2], [3], [5], [6], [11] to isolate specific microarchitectural effects and narrow the scope of analysis. More recent calibration work, Cliff Microbenchmarking [13] emphasizes single-feature attribution by constructing targeted microbenchmarks that exercise one microarchitectural feature at a time, enabling direct exposure of behavioral discrepancies at a fine granularity.

Although a broad set of features can be calibrated using strategies similar to those described in the Cliff methodology [13], generating these calibration-oriented microbenchmarks still requires substantial manual development of core program logic. This process requires significant domain expertise and is often tedious and error-prone. As modern microarchitectures continue to pursue higher customization and workload-specific specialization [7], the number of microarchitectural features that require calibration also increases, posing challenges to the rapid iteration of model calibration.

The construction of calibration-oriented microbenchmarks primarily requires strict adherence to predefined design rules. Such rule-driven tasks are particularly well suited to large language models (LLMs), which can systematically follow structured constraints to generate compliant benchmarks. Given expert-specified benchmark construction rules, LLMs can be leveraged to automatically generate calibration-oriented microbenchmarks, significantly improving the efficiency of the calibration workflow.

Based on these observations, we propose *AC Loop*, an LLM-driven end-to-end automated calibration framework. AC Loop is capable of checking and automatically calibrating configuration parameters between the simulator and the RTL implementation. AC Loop takes as input a human-written description of the feature under evaluation and the specific calibration task, automatically synthesizes targeted microbenchmarks, and iteratively refines the microbenchmarks based on execution feedback. To facilitate the workflow, we developed a code generation tool, referred to as the *Code Generator* (CG), that encapsulates a rich set of abstractions for calibration-oriented microbenchmark construction, which can be automatically invoked by the LLM as a subroutine. By providing the LLM with sufficient background knowledge and structured guidance, it can assist in automatically invoking and modifying CGs and iteratively refining the benchmark generation strategy based on checkpoint feedback, thereby ensuring the correctness and reliability of the generated code.

We evaluate AC Loop on the XiangShan version of gem5 (XS-GEM5) and the XiangShan open-source CPU (XS-RTL) [8]–[10], [12]. Experimental results show that AC Loop can effectively correct misconfigured parameters, generate calibration-oriented benchmarks, accurately assess calibration

quality between the simulator and the RTL implementation, and identify fine-grained root causes of mismatches. We also evaluate the computational overhead of the LLM-based benchmark generation process. In the case study on load queue size related benchmark generation, the model consumed 7,188 input tokens and generated 3,568 output tokens. The total code generation latency was 464.6 seconds.

II. Microbenchmarking

Microbenchmarking has become a primary technique for calibrating a performance model with the target hardware. Early work [4] highlighted the existence and potential impact of modeling errors and emphasized performance model calibration as a first-class goal in design space exploration. They proposed an iterative calibration process using synthetic test cases of increasing complexity to gradually expose and correct modeling mismatches. Subsequent works on calibrating performance models [2], [3], [5], [6], [11] have largely relied on carefully designed microbenchmarks to exercise specific processor behaviors, such as branch prediction, execution and forwarding latencies, and memory access bandwidth across different levels of the memory hierarchy. Common analysis techniques include leveraging available public documentation, performance counter analysis, and inferring from empirical observations.

A step closer to an automatic calibration framework is presented in [1], where a set of microbenchmarks and machine-learning-based search are employed to tune simulator parameters until the hardware behavior is matched.

Across these prior works, we observe that the quality and representativeness of microbenchmarks are critical to ensuring the effectiveness of these calibration methods. However, most existing microbenchmarks are manually crafted by domain experts, which increasingly becomes a bottleneck for scalable and systematic calibration as microarchitectural designs continue to evolve.

III. Motivation

To ensure simulation accuracy, model calibration is typically performed to align the simulator with the RTL implementation at the architectural level. However, existing performance calibration methods typically rely on microbenchmarks during the calibration process [13]. Modern processor microarchitectures comprise numerous intricate microarchitectural features, making it necessary to construct a large number of microbenchmarks for comprehensive calibration. Meanwhile, manually developing such microbenchmarks is labor-intensive and requires substantial engineering effort. In addition, manual construction should strictly adhere to numerous low-level rules. For example, (1) real RAW (Read-After-Write) dependencies should be explicitly enforced; (2) cache warm-up effects should be precisely controlled to affect only designated execution phases. Such manual construction is tedious and error-prone.

The purpose of microbenchmark construction is to support microarchitectural calibration and to evaluate its effectiveness. In practise, however, manual microbenchmark development itself incurs substantial engineering effort during the calibration process. Notably, these microbenchmarks construction is largely governed by strict rule compliance and mechanical correctness rather than creative reasoning. This rule-driven nature makes the process well-suited for automation.

Recent advances in large language models (LLMs), which excel at following structured instructions and generating syntactically correct code, provide a promising opportunity to automate benchmark generation [14]. Moreover, because calibration requires single-feature attribution [13], the task naturally decomposes into well-defined, isolated objectives, enabling LLMs to focus on satisfying explicit constraints for a specific microarchitectural feature and thereby improving generation reliability. When guided by explicit specifications, structured workflows, and predefined validation checkpoints designed to verify rule compliance, LLMs can reliably generate benchmarks consistent with calibration requirements. In this framework, domain experts define calibration rules, the LLM synthesizes benchmark code, and the broader community can execute experiments and evaluate calibration results.

More importantly, such automation substantially reduces human effort at scale. With clearly specified rules and properly configured validation checkpoints, LLMs are well positioned to perform repetitive and rule-dense benchmark generation more reliably than manual development. By leveraging a systematic and fully automated framework, experienced experts can focus their expertise on simulator calibration and performance analysis, rather than on repetitive benchmark construction tasks.

IV. Design

To better leverage LLMs to improve the efficiency of microbenchmark generation and the overall calibration workflow, we face four key challenges. First, how can we help the LLM better understand our codebase and the relevant microarchitectural context? Second, how can we guide the LLM to generate appropriate calibration-oriented microbenchmarks? Third, how should we process and analyze the resulting data to diagnose calibration gaps? Finally, once discrepancies are detected, how can we translate them into actionable fixes by modifying the simulator in a controlled manner?

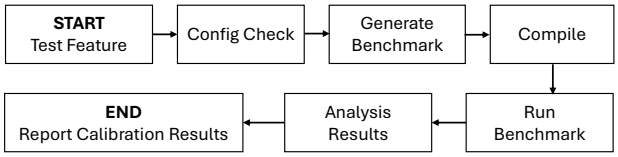

Fig. 1. LLM-Based End-to-End Auto Calibration Framework.

To address the challenges above, we first abstract microbenchmarks into a Python-based representation, which makes the code easier for the LLM to understand and generate correctly. In addition, as shown in Fig. 1, we propose an automated end-to-end calibration framework that systematically

orchestrates the entire workflow, including parameter configuration validation and calibration, LLM-based microbenchmark generation, execution, result analysis, and calibration reporting.

A. Helping Large Language Models Better Understand Microbenchmarks

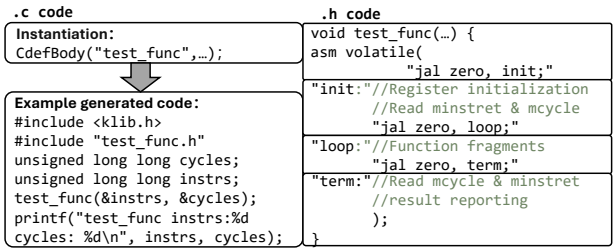

Fig. 2. Example of generated .c and .h code.

Our analysis of existing calibration microbenchmarks shows that their code structures follow highly consistent patterns and offer significant opportunities for reuse. Each microbenchmark typically consists of a .c file and a .h file, as shown in Fig. 2. The .h file contains header inclusions, macro definitions, cache warmup functions, and the actual benchmark function. The .c file is responsible for including headers, invoking the benchmark function, handling input/output parameters, and printing the result report. Most of the remaining code is highly reusable and follows a uniform structure.

For the code generator, users only need to write a top-level Python driver that invokes the generator, referred to as the Code Generator (CG), and provide concise test code snippets. The tool then automatically expands these snippets into complete and executable microbenchmark programs. The code generator provides multiple layers of code abstraction, including main-function generation, test-function definition, initialization routines, instruction-snippet encapsulation, and result-reporting abstraction. These modular abstractions significantly simplify the development process, enabling developers to efficiently and conveniently construct large-scale microbenchmarks for calibration. Users only need to modify the input instruction snippets, making the microbenchmark generation process work out-of-the-box. By abstracting structural complexity, CG not only lowers the barrier for developers to write microbenchmarks but also facilitates LLM-based microbenchmark generation, guiding LLMs to produce microbenchmarks that conform to the required specifications.

B. Automatic Detection and Correction of Configuration Errors

As part of the end-to-end framework, we automatically detect parameter configuration mismatches and apply corresponding corrections. At the beginning of the calibration process, we check whether the relevant configuration parameters of the simulator and the RTL are consistent. If inconsistencies are found, scripts are used to modify the simulator configuration to match the RTL. After the configuration is updated,

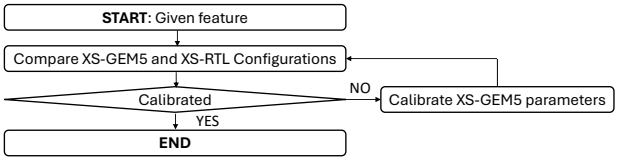

Fig. 3. Automated Configuration Checking and Calibration Workflow.

the workflow proceeds to microbenchmark generation only if the consistency check passes. If the mismatch persists or the correction fails, the framework terminates and requires manual configuration inspection.

C. Guiding LLM to Operate as Intended

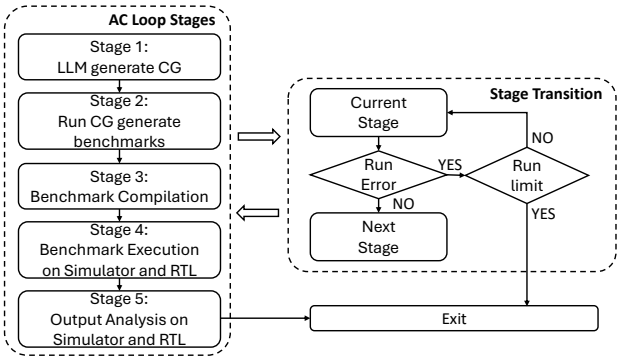

Fig. 4. LLM-Based Code Generation Workflow.

To ensure that the LLM produces microbenchmarks that correctly evaluate performance calibration between the simulator and the RTL, we decompose the workflow into a sequence of well-defined subtasks and enforce validation checkpoints at each stage. The overall workflow is illustrated in Fig. 4. When a subtask fails to satisfy the expected criteria, the execution context and the target outcome are fed back to the LLM for iterative refinement. To guarantee system robustness and avoid deadlock in the automated pipeline, we impose a strict upper bound on the number of retries for each subtask. If the checkpoint is not satisfied after exceeding this limit, the system reports an error and terminates the current run, deferring the resolution to manual intervention.

Specifically, we provide the LLM with essential background knowledge and instruct it to generate a microbenchmark code generator (CG) according to the given requirements. The code generator is implemented in Python and abstracts program behavior at a higher level, enabling the LLM to better understand code semantics and generate corresponding benchmarks more effectively. After the LLM generates the code generator, the generator is executed automatically. If syntax errors occur, the error context together with the generated code is fed back to the LLM, which is then instructed to revise and correct the implementation.

Since the simulator's parameter configuration is known, we have a reference "ground truth" for performance evaluation. If the performance results from both the simulator and the

RTL deviate significantly from the expected RTL behavior, we conclude that the benchmark fails to correctly capture the performance bottleneck. In this case, both the execution results and the benchmark code are fed back to the LLM for revision.

D. Data Processing and Analysis

To analyze the results in a more systematic and detailed manner, we categorize them into three types. (1) Trendline-based analysis. As the number of tested instructions increases, IPC rises monotonically and then saturates at a peak, indicating the maximum bandwidth of the corresponding functional unit. (2) Change-point detection. We introduce backpressure with long-latency instructions and progressively fill the target resource with short-latency instructions. A sharp latency jump occurs once outstanding short-latency instructions exceed a threshold, indicating resource saturation and thus maximum capacity. (3) Comparative performance analysis. We construct comparison experiments by enabling or disabling specific trigger conditions. Comparing performance under triggered and non-triggered scenarios allows us to validate the presence and impact of the corresponding microarchitectural feature.

V. Evaluation

To evaluate the feasibility of the automated workflow, we assess the entire pipeline from two perspectives. First, we evaluate whether the code generator can correctly produce microbenchmark code. Second, we evaluate whether the framework can effectively accomplish its intended tasks, including leveraging LLM to generate codes and perform calibration-oriented performance analysis.

A. Experiment Setup

Platforms. Using AC Loop, we curate a set of microbenchmarks to support calibration between XS-GEM5 and XS-RTL [8]–[10], [12].

LLMs. The proposed framework is validated on state-of-the-art (SOTA) large language models, including Claude 4.5 Opus.

B. Evaluation of the Code Generation Tool

We develop a code generator (CG) to automate microbenchmark construction. Given only instruction snippets, it generates complete microbenchmark programs. With the support of the CG, the LLM focuses solely on generating the core test snippets, improving both reliability and efficiency. Taking the LoadQueueSize test as an example, approximately 200 lines of Python code are sufficient to generate 6,332 lines of complete microbenchmark programs. In addition, we implement a fully automated workflow using scripts to handle compilation, execution, and data collection, greatly improving testing efficiency and reproducibility.

C. End-to-End Workflow Evaluation

We take the Load Queue Size microbenchmark generation as a case study to validate the AC-calibration framework. The experimental results show that AC-calibration can fix incorrect parameter configurations, generate valid performance benchmarks, and report the calibration status. In particular,

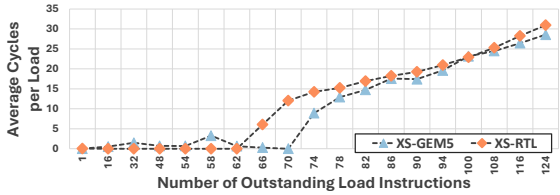

Fig. 5. benchmark of load queue size.

when XS-GEM5 is misconfigured with an incorrect load queue size (initialized to 128 entries), the AC-calibration framework successfully detects the configuration mismatch and corrects the parameter accordingly.

For microbenchmark generation, we executed the AC-calibration workflow ten times. Although the generated CG comments varied across runs, each attempt successfully produced correct CG code. As shown in Fig. 5, using benchmarks generated by the AC-calibration framework, we are able to correctly evaluate the calibration between XS-GEM5 and XS-RTL. We further apply Python scripts to fit and analyze the results from XS-GEM5 and XS-RTL, examining change points and overall trend behaviors. We demonstrate the capability of the AC calibration framework by highlighting both well-aligned behaviors and regions of misalignment between the simulator and the RTL implementation.

1) Consistent Behaviors: We observe consistent performance trends in two regions: when the number of outstanding load instructions is below 62 and when it exceeds 78. Given that the ground-truth Load Queue capacity is 72 entries, this indicates that the simulator and RTL implementation exhibit aligned behavior under both low and high occupancies. In contrast, discrepancies emerge when the outstanding load demand is near the capacity boundary. Next, we provide analysis on the misaligned regions.

2) Identifying Mismatches: As the generated microbenchmark has already identified a key region in which the simulator diverges from the RTL behavior, debugging becomes substantially more direct. The discrepancy arises when the Load Queue frequently oscillates between ”full” and ”not full”. A quick inspection reveals that the RTL asserts the Load Queue as full when the number of free entries fall below the maximum enqueue bandwidth. Concretely, with an enqueue bandwidth of 6 loads per cycle and a 72-entry Load Queue, the RTL treats the queue as effectively full when the occupancy is between 66 to 72 entries. This policy is not applicable to the gem5 model and gem5 continues allocating load entries until all 72 entries are fully occupied. As a result, we see a later performance inflection point for the simulator than for the RTL. Overall, AC Loop successfully generates targeted benchmarks that capture the full state space of microarchitectural responses, enabling focused and systematic analysis to precisely root causes.

3) Cost Analysis: Table I summarizes the token usage and latency statistics of the LLM-based generation process. We evaluate the computational overhead of the LLM-based

TABLE I
LLM GENERATION STATISTICS

Metric	Value
Input Tokens	7188
Output Tokens	3568
Total Latency	464.6 s
Approx Output Throughput	7.68 tokens/s

benchmark generation process. In a representative run, the model consumed 7,188 input tokens and generated 3,568 output tokens. The total end-to-end latency was 464.6 seconds, while local postprocessing introduced negligible overhead (0.1 ms). The observed output throughput was approximately 7.68 tokens per second. Although the majority of the end-to-end latency arises from LLM inference, this overhead is acceptable in practice. Compared to manual benchmark development, which typically requires hours of engineering effort, the automated generation process offers a practical and scalable alternative.

VI. CONCLUSION

We present AC Loop, an LLM-based end-to-end automated calibration framework. AC Loop effectively automates the calibration of configuration parameters and leverages large language models to construct performance-calibration benchmarks for evaluating the consistency between the simulator and the RTL. We evaluate AC Loop on XS-GEM5 and XS-RTL. Experimental results show that, AC Loop can effectively identify and correct misconfigured parameters, generate appropriate performance benchmarks, and report the calibration status between the simulator and the RTL.

REFERENCES

[1] A. Adileh, C. González-Álvarez, J. M. D. H. Ruiz, and L. Eeckhout, "Racing to hardware-validated simulation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 58–67.

[2] A. Akram and L. Sawalha, "Validation of the gem5 simulator for x86 architectures," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019, pp. 53–58.

[3] M. A. Z. Alves, C. Villavieja, M. Diener, F. B. Moreira, and P. O. A. Navaux, "Sinuca: A validated micro-architecture simulator," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, 2015, pp. 605–610.

[4] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *Computer*, vol. 31, no. 5, pp. 59–65, 1998.

[5] R. Desikan, D. Burger, and S. W. Keckler, "Measuring experimental error in microprocessor simulation," in *Proceedings of the 28th annual international symposium on Computer architecture*, 2001, pp. 266–277.

[6] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 13–22.

[7] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[8] W. Kaifan, X. Yinan, Y. Zihao, T. Dan, C. Guokai, C. Xi, G. Lingrui, H. Xuan, J. Yue, L. Qianruo, L. Xin, L. Jiawei, L. Tong, L. Zhigang, W. Huaqiang, W. Huizhe, Z. Chuanqi, Z. Fawang, Z. Linjuan, Z. Zifei, Z. Ziyue, Z. Yangyang, Z. Yaoyang, Z. Jiangrui, C. Ye, H. Dandan, L. Zusong, Z. Jiye, H. Wei, S. Ninghui, and B. Yungang, "Xiangshan open-source high performance risc-v processor design and implementation," *Journal of Computer Research and Development*, vol. 60, no. 3, pp. 476–493, 2023. [Online]. Available: https://crad.ict.ac.cn/en/article/doi/10.7544/issn1000-1239.202221036

[9] OpenXiangShan, "Xiangshan," https://github.com/OpenXiangShan/XiangShan, 2025.

[10] OpenXiangShan, "xiangshan-gem5," https://github.com/OpenXiangShan/GEM5, 2025.

[11] R. Singhal, K. Venkatraman, E. R. Cohn, J. G. Holm, D. A. Koufaty, M. J. Madhav, M. Mattwandel, N. Nidhi, J. D. Pearce, M. Seshadri *et al.*, "Performance analysis and validation of the intel® pentium® 4 processor on 90nm technology." *Intel Technology Journal*, vol. 8, no. 1, 2004.

[12] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li *et al.*, "Towards developing high performance risc-v processors using agile methodology," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1178–1199.

[13] H. Zhen, Q. Kang, Y. Bao, and T. E. Carlson, "Benchmarking for single feature attribution with microarchitecture cliffs," 2026. [Online]. Available: https://arxiv.org/abs/2602.11580

[14] Q. Zhou, Y. Wen, R. Chen, K. Gao, W. Xiong, L. Li, Q. Guo, Y. Wu, and Y. Chen, "Qimeng-gemm: Automatically generating high-performance matrix multiplication code by exploiting large language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 21, 2025, pp. 22 982–22 990.