LLM-assisted Performance Estimation of Embedded Software on RISC-V Processors

Weiyan Zhang[1] Muhammad Hassan[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{weiyan, hassan, drechsler}@uni-bremen.de

***Abstract*—In this paper, we present a methodology that combines a *Large Language Model* (LLM) with a traditional *Machine Learning* (ML) approach to estimate the performance of embedded software on RISC-V processors across different microarchitectures. In particular, we employ a *Retrieval-Augmented Generation* (RAG)-based LLM to extract performance-related information from processor specifications and source code. Additionally, we leverage the predictive capabilities of ML models to create *Predictive Models* (PMs) for RISC-V processors. To demonstrate the effectiveness of our hybrid approach, we present results on the performance estimation of open-source benchmarks using the generated PMs, with open-source RISC-V-based *Register Transfer Level* (RTL) implementations as reference models. Our results demonstrate that our proposed LLM-assisted methodology provides highly accurate predictions, with *Mean Absolute Percentage Errors* (MAPEs) of only 2.50% for SweRV core and 11.90% for RSD core, respectively. In comparison with the state-of-the-art methodology, our approach achieves significant improvements, reducing the MAPE by 61.54% for SweRV and 36.02% for RSD.**

***Index Terms*—RISC-V, performance estimation, large language model, machine learning, embedded system**

I. Introduction

Embedded systems, which are becoming more complex as silicon technology advances and their functionality grows, have become increasingly important in automation and the *Internet of Things* (IoT) in recent years. RISC-V [1] has gained popularity in this field due to its open-source architecture, which allows developers to customize designs to meet specific requirements while keeping costs low and avoiding licensing restrictions. Companies such as SiFive [2] have leveraged RISC-V to develop customizable and energy-efficient processors, demonstrating its real-world applicability in IoT devices, edge computing, and *Artificial Intelligence* (AI) accelerators. Additionally, RISC-V has a comprehensive and expanding ecosystem [3], including tools such as functional simulators and *Register Transfer Level* (RTL) implementations. Those various RISC-V implementations across different levels of abstraction make it adaptable to a wide range of applications, allowing for both high-level customization and low-level optimization to meet specific performance, power, and cost requirements.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, and EXCL under grant no. 01IW22002.

979-8-3315-2801-0/25/$31.00 ©2025 IEEE

Performance estimation, especially in terms of cycle count, is critical for system design as it helps ensure that systems meet performance requirements. However, as embedded systems grow more complex, achieving accurate performance estimation becomes more challenging. The difficulty lies not only in managing the complexity of these systems, but also in accounting for variations in system behavior across different use cases. Traditional methods like RTL simulations are precise but slow, which can delay design decisions and affect time-to-market goals. This creates a trade-off between accuracy and speed, making efficient performance estimation crucial for staying competitive in fast-paced industries.

Recent advancements in AI, especially in the area of *Large Language Models* (LLMs), have expanded their applications beyond traditional natural language processing, now playing new roles in the development of embedded systems. Models like OpenAI's GPT series demonstrate strong capabilities in understanding and processing both natural [4] and programming languages [5], making them highly effective for interpreting and translating hardware specifications and source code. LLMs are now being used to assist in various stages of embedded systems development, particularly in areas such as design automation, performance optimization, and verification. *Retrieval-Augmented Generation* (RAG) techniques can further enhance the capabilities of LLMs by enabling them to pull relevant external data, allowing for more contextually accurate responses. By leveraging the large amount of data and documentation available, LLMs can help engineers develop and debug embedded systems more efficiently. They can automatically extract performance-related information from technical documents and source code, enabling the generation of accurate *Predictive Models* (PMs) of hardware. This accelerates tasks like performance estimation of embedded software on specific processors, reducing time-to-market pressures and improving the overall efficiency of embedded systems design workflows. However, LLMs have limitations when handling tasks that require precise calculations or numerical reasoning [6]. Traditional *Machine Learning* (ML) techniques may help address this issue, as many statistical models, such as those proposed in [7], [8], have demonstrated good results in certain aspects of performance estimation. To further overcome these limitations, traditional ML models, such as *Artificial Neural Networks* (ANNs), can be integrated to provide more accurate solutions for tasks that involve complex calculations.

Contribution: In this paper, we propose leveraging state-

of-the-art LLM to extract performance-related information for RISC-V processors and using ML models to train the PMs. The methodology comprises four phases: text generation, data preprocessing and RAG, model training, and model testing. We used around 700 programs to train the PMs by extracting execution metrics from two RISC-V RTL implementations, SweRV [9] and RSD [10], as well as the functional simulator Whisper [11]. For testing, 10 distinct benchmarks from TACLeBench [12] were used to evaluate the PMs' accuracy by comparing predicted clock cycles to ground truth values from RTL simulations. The results show *Mean Absolute Percentage Errors* (MAPEs) of 2.50% for SweRV and 11.90% for RSD, demonstrating the method's high accuracy and effectiveness, even without physical hardware.

II. Related Work

Fast and accurate performance estimation is essential in embedded systems, as it plays a key role in optimizing system design and resource allocation. To address this need, various methods have been developed to improve the efficiency and accuracy of performance prediction. In this section, we review the most relevant state-of-the-art performance estimation methods related to our work.

In general, performance estimation methods can be divided into three main categories:simulation, analytical, and statistical.

Simulation-based methods use cycle-accurate or cycle-approximate simulators. Cycle-accurate simulators, like RTL simulators [13], offer detailed and precise insights into a system's behavior but come with significant runtime overhead [14]. On the other hand, cycle-approximate simulators, such as gem5 [15], can achieve faster execution but compromise on accuracy, making them less reliable for detailed performance analysis. High-level abstract simulations, including those utilizing SystemC-*Transaction Level Modeling* (TLM), enable faster simulations by abstracting lower-level details like individual instruction timings. RISCV-VP [16] and RISCV-TLM [17] are RISC-V simulators built on SystemC-TLM, incorporating an ISS, memory, and a basic set of peripherals.

Analytical methods, such as those used in static profilers [18], estimate performance through mathematical models without the need to execute the code. However, they cannot capture runtime behavior. In recent research, Prof5 [19] employs a hybrid analysis approach for RISC-V designs, combining fast functional simulation using Spike [20] with energy and timing models derived from RTL simulations, achieving an $8000\times$ speed-up with an average accuracy of 95%. RV-ProViler [21] integrates the RISC-V GNU toolchain [22] and the Spike simulator using four scripts to generate a histogram of instruction coverage, helping to identify the instructions required for specific applications or functions. To demonstrate its feasibility, various functions across different RISC-V specifications were evaluated, achieving a maximum speed-up of $5.87\times$ with the P-extension and $16.52\times$ with the V-extension.

Statistical methods use performance counters to model the relationship between embedded software and run-time profiling. Traditional statistical models are typically designed to work in cases where the relationship between input and output data is not easily defined by a simple equation or statistical method, such as performance estimation using linear regression [23], [24]. More advanced statistical models leverage complex ML algorithms, such as ANN, to enable models to learn from input data and make accurate predictions. ANN-based methods effectively adapt to non-linear behaviors like pipelines, branch prediction, and caches, while offering fast estimation speeds. For instance, [25] used an ANN to predict instruction throughput based on the opcodes and operands of instructions in a basic block, demonstrating higher accuracy compared to state-of-the-art analytical models. [8] developed an ML-based estimation model using dynamic instruction counts for the RISC-V processors. However, modeling complex architectures remains challenging due to the lack of sufficient performance-related features provided by the processor.

In the last three years, the capabilities of LLMs, such as GPT-3, GPT-4, and Gemini, have significantly advanced due to the rapid increase in model sizes and the availability of extensive training datasets. This evolution brings both challenges and opportunities in the field of hardware design [26]. Because of their natural language processing ability, LLMs have expanded their applications across various stages of the design process, including verification [27], code generation [28], and SoC security [29]. However, the computational demands associated with LLMs present certain challenges [6], such as those encountered in mathematical reasoning. To address these challenges, we first utilize an LLM to process natural language documentation and source code, extracting performance-related information. Subsequently, we employ an ANN to perform performance estimation based on the extracted data. This integrated approach enables us to leverage the LLM's strengths in understanding complex text while utilizing the ANN for efficient computational tasks, ultimately enhancing the accuracy and efficiency of our performance assessments.

III. LLM-Assisted Methodology for Performance Estimation

Our LLM-assisted methodology for performance estimation of embedded software, illustrated in Fig. 1, integrates ANN and consists of four phases: text generation, data preprocessing and RAG, model training, and model testing. Each phase is detailed in the following subsections.

A. Text Generation

In the text generation phase, an LLM is employed to assist in identifying all potential parameters that could impact the performance of the system, particularly those related to the number of execution cycles. The process starts with a designed prompt that provides initial examples, such as *multiplier_latency* and *branch_hit_penalty*, and instructs the LLM to extract additional relevant parameters that can be extracted from RTL source code and corresponding documentation. The resulting parameter list is used in subsequent steps to extract values for each RTL core.

B. Data Preprocessing and RAG

This phase has two main components: data preprocessing, which creates a vector database, and RAG, which retrieves the required information.

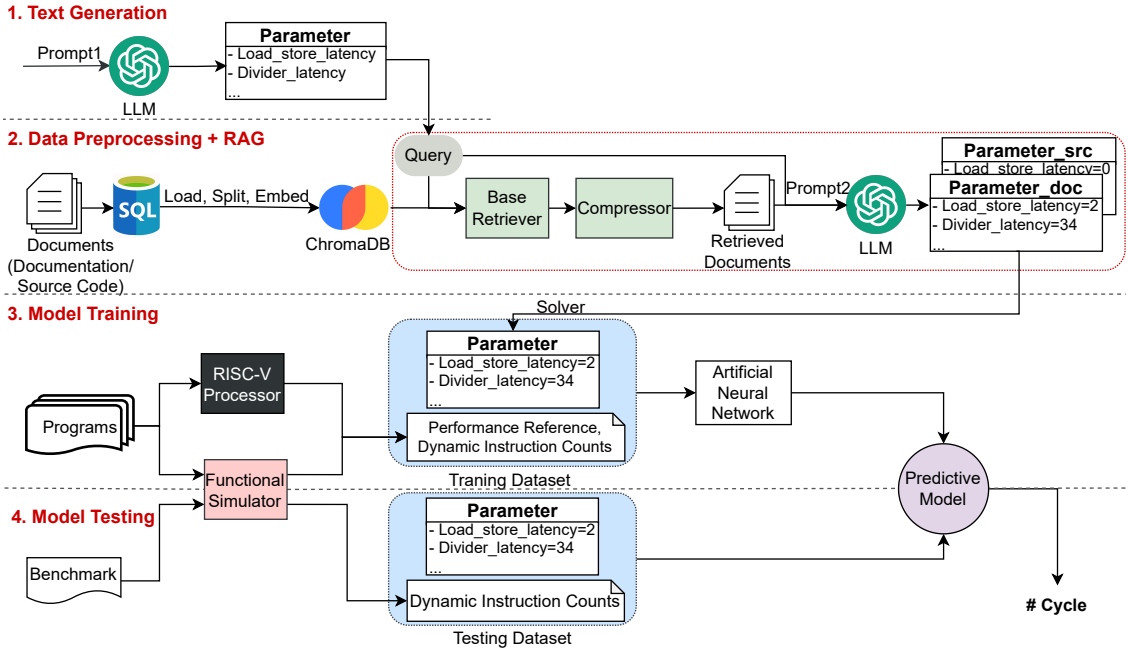

Fig. 1. Overview of the proposed LLM-assisted methodology for performance estimation of embedded software.

In this phase, documentation and source code are processed in parallel but handled separately. For documentation, preprocessing collects relevant files, stores them in an SQL database, and splits them into smaller chunks for efficient searching within the LLM's context window. Each chunk is embedded into a vector representation and stored in a vector database, such as ChromaDB [30], for retrieval. The RAG component queries stored chunks to extract parameter values. When a query is issued, such as asking for a specific parameter, the system retrieves relevant chunks from the vector database using a retriever. A compressor reranks the results to filter out irrelevant content, refining the search. The refined results are then passed to an LLM, which, guided by a prompt, generates a list of extracted parameter values from the documentation.

The same process is applied to the source code, where it is preprocessed, split into chunks, embedded into vectors, and stored. Using the same query, the system retrieves, ranks, and processes the relevant code splits to produce a list of parameter values.

By the end of this phase, two lists are generated: one from the documentation and the other from the source code, both essential for the subsequent model training stages.

C. Model Training

In the model training phase, programs are executed on a RISC-V processor and a functional simulator to obtain reference clock cycles and dynamic instruction counts, respectively. Dynamic instruction counts refer to the number of each RISC-V instruction required for program execution. Since real hardware may not be available for testing, the RISC-V RTL implementation is used as a substitute for generating these metrics. A conflict-resolution solver addresses inconsistencies in parameter lists extracted from documentation and source code, where the same parameter may have different values. For example, the load_store_latency might be 2 in the documentation but 0 in the source code. In such cases, the solver prioritizes the non-zero value. If both values are non-zero, the solver prioritizes the value from the source code as more reliable, producing a single, unified parameter list. An ANN models the relationship between parameters and instruction counts as inputs, and the number of clock cycles as outputs, enabling accurate predictions of processor behavior.

D. Model Testing

To evaluate the accuracy and effectiveness of the PM, the number of executed clock cycles required by the new software is predicted. The new software is selected from standard benchmark suites to ensure a diverse range of test cases. The software is executed on a functional simulator to obtain dynamic instruction counts, which, along with performance-related parameters obtained using LLM, are provided to the PM. The PM predicts the clock cycles needed for execution, and its accuracy is assessed by comparing this prediction with the actual execution cycles.

IV. Experimental Evaluation

In this section, we present the experiments to demonstrate the effectiveness of our proposed performance estimation methodology.

A. Experimental Setup

The LLM used in the first two phases is gpt-3.5-turbo-0125 [31], accessed via OpenAI's *Application Programming Interface*(API). An SQL database was used to collect and organize source code and documentation, enabling structured queries and better data management. After loading the content from the SQL database, we leverage LangChain's framework [32] to connect LLMs with the data sources and services throughout the following steps of this phase. The CharacterTextSplitter [33] was applied to split the text into smaller

chunks based only on a specified separator. In this experiment, the default separator "\n\n" was used, and the chunk size was limited to 1000 characters to balance processing efficiency and context retention. These chunks were embedded into vector representations using the all-MiniLM-L6-v2 model [34] from sentence-transformers and stored in a Chroma database for fast, vector-based searches. In the RAG component, a vector store-backed retriever [35] searched the vector representations of document chunks in the Chroma database to retrieve those most similar to a given query. FlashrankRerank [36] was applied as a compressor to filter out less relevant chunks and rerank the top 10, prioritizing the most relevant information. Finally, gpt-3.5-turbo-0125 extracted performance-related parameters from the reranked retrieved chunks.

During the model training phase, we used a set of around 700 programs, generated by varying inputs to custom-written sample programs as well as incorporating several standard benchmarks from TACLeBench [12]. These programs were compiled with the RISC-V GNU toolchain [22] for RV32I and executed on two RISC-V RTL implementations, SweRV [9] and RSD [10], to collect reference clock cycles, as physical hardware was unavailable. Dynamic instruction counts were gathered using the functional simulator Whisper [11]. The execution metrics and performance-related parameters were combined to form the training dataset. The ANN was implemented in TensorFlow 2.6.0 [37], with hyperparameters such as learning rate, number of hidden layers, neurons per layer, activation functions, and epochs defined in a comprehensive search space. A random search technique [38] was employed to efficiently tune and select optimal hyperparameters by randomly sampling values.

During the model testing phase, we selected 10 benchmarks from TACLeBench, distinct from the training set, spanning domains like signal processing and mathematical problem-solving. These benchmarks, designed for embedded systems, are freely accessible. Dynamic instruction counts were collected by executing the software on the Whisper functional simulator.

Python 3.9 was used to implement both the LLM and ANN components. While GPT-3.5-turbo-0125, accessed via the OpenAI API, is not open-source, all other tools — including the LangChain, TensorFlow, Whisper, the RISC-V implementations, and TACLeBench — are open-source and freely available, ensuring the workflow is replicable for the research community.

B. LLM Implementation Details

Fig. 2 provides an overview of the interaction between the user and the LLM during the first two phases of extracting performance-related parameters from RTL documentation and source code.

In phase 1, the user queries GPT-3.5-turbo-0125 for key performance-related parameters, such as *multiplier_latency*, *divider_latency*, and *load_store_latency*, which directly impact processor performance. The LLM, using its pre-trained knowledge, identifies these parameters from the RTL documentation and source code.

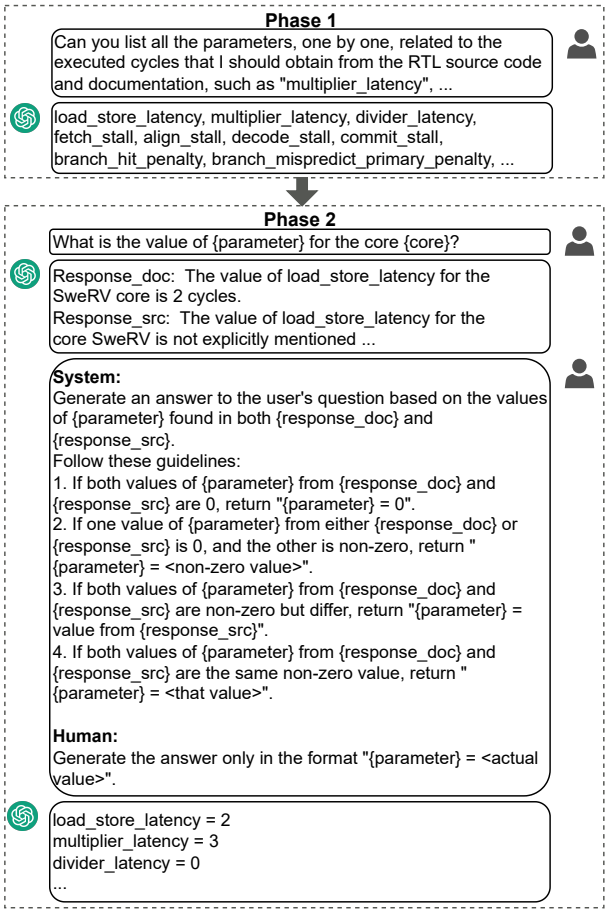

Fig. 2. Illustration of user-LLM interactions during phases 1 and 2 for extracting performance-related parameters from RTL documentation and source code.

Phase 2 extracts numerical values by dividing the task into sub-queries, each focusing on a specific parameter. Using a RAG approach, the LLM retrieves relevant document chunks and extracts values step by step for improved accuracy.

The extracted values from the RTL documentation and the source code may differ due to discrepancies or incomplete information in one source compared to the other. Discrepancies between values from documentation and source code are resolved using a conflict-resolution strategy guided by a ChatPromptTemplate [39], which prioritizes source code values as more reliable. For instance, if the value of a parameter from the documentation is 0 while the source code provides a non-zero value, the LLM is instructed to prioritize the non-zero value. Similarly, if both values are non-zero but differ, the value from the source code is preferred, as it is considered a more reliable reflection of the actual hardware behavior. These conflict-resolution rules ensure consistency and reliability in the extracted data. Additionally, to improve the clarity and readability of the LLM's responses, the user specifies a strict output format for the LLM to follow. For clarity and consistency, the LLM follows a strict output format, presenting results as "parameter = value", ensuring they are interpretable and ready for analysis. By structuring the process into phases and stepwise extractions, this methodology achieves high accuracy and efficiency.

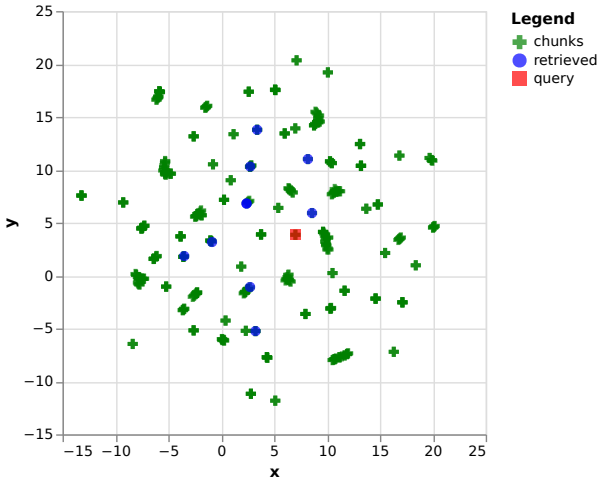

Fig. 3. UMAP visualization of the retrieved document chunks for the query "What is the value of divider_latency for the core SweRV?".

To visualize the RAG process in phase 2, we use *Uniform Manifold Approximation and Projection* (UMAP) [40], which provides a clear overview of the data by reducing the high-dimensional embeddings into a 2D space. This reduction allows for easier interpretation and analysis while maintaining key properties such as the relationships and proximities between the document chunks and the query. By preserving these spatial relationships, UMAP enables a visual assessment of how well the retrieved chunks align with the query, offering insights into the effectiveness of the retrieval process.

Fig. 3 shows an example of the 2D projection of high-dimensional embeddings, where the proximity of chunks to the query represents their relevance to the requested information. In this visualization, the red square represents the query "What is the value of divider_latency for the core SweRV?". The blue circles correspond to the 10 retrieved chunks, which are considered most relevant to the query after being processed by the retriever and reranked by the compressor. The green crosses represent all the chunks from the documentation, providing a broader context of the entire database. During the experiment, the UMAP plot was generated in .svg format, providing an interactive experience. In this format, the content of each point in the UMAP plot, whether represented by dots, crosses, or square, can be directly inspected. This interactivity allows for a detailed examination of the retrieved chunks, enabling us to manually assess their relevance to the query.

C. Performance of RISC-V Predictive Models

To evaluate the effectiveness of the generated PMs, we assessed their accuracy against the corresponding RISC-V RTL implementations, SweRV and RSD. The results, presented in Table I, provide the *number of executed instructions* (# instr-exec.), the *Source Lines of Code* (SLOC), and the *Absolute Percentage Error* (APE) for each individual benchmark, which measures the magnitude of the error as a percentage, along with the *Mean Absolute Percentage Error*(MAPE) calculated across all test benchmarks. This evaluation allows us to assess how closely the PMs estimate the number of executed cycles compared to the ground truth provided by the RTL simulations.

TABLE I
EXPERIMENTAL RESULTS OF ALL BENCHMARKS USED FOR VALIDATION OF PMS ON RV32I

benchmark	# instr-exec.	SLOC	**SweRV** # Cycle	**PM1** APE	**RSD** # Cycle	**PM2** APE
adpcm_dec	2 880 766	397	3 644 041	0.21%	5 260 228	0.87%
adpcm_enc	2 898 909	413	3 224 628	13.59%	5 232 514	0.24%
cubic	28 338 773	646	34 071 398	1.34%	64221227	23.68%
deg2rad	510 731	32	573 745	2.51%	872 615	1.12%
fft	3 678 522	492	5 420 220	1.15%	4 010 734	43.26%
gsm_dec	9 168 156	504	14 387 983	2.50%	12 076 100	30.08%
isqrt	1 002 078	629	3 125 021	1.30%	1 220 580	0.72%
lms	5 814 943	114	7 063 929	0.79%	10 253 500	1.04%
rad2deg	420 103	32	482 789	0.51%	627 494	10.17%
st	3 684 066	127	4 445 458	1.05%	5 842 673	7.88%
MAPE				2.50%		11.90%

As seen in the results, the PMs exhibit varying levels of accuracy across the different benchmarks, with PM1 generally achieving lower APEs than PM2 for most benchmarks. For instance, the MAPE for PM1 is 2.50%, while PM2 shows a higher MAPE of 11.90%. This suggests that PM1 is better optimized for the SweRV, while the lower accuracy of PM2's performance estimation for the RSD can be attributed not only to the need for further tuning but also to the quality and completeness of the documentation, which may have hindered the LLM's ability to extract accurate parameters. When analyzing the APE for each benchmark, we observe that PM1 performs well across most benchmarks, except for adpcm_enc, where the APE rises to 13.59%. This higher error, along with PM2's inconsistent performance — particularly in benchmarks like fft (43.26%) and gsm_dec (30.08%) — is likely due to the complexities in certain benchmarks, such as irregular memory access and dynamic execution patterns, which are harder to model accurately. The differences in PM2's performance reflect the architectural complexities of the RSD and the challenges in accurately modeling these behaviors using the extracted performance-related parameters and instruction counts.

The generation of the training dataset for model training involved contributions from the LLM, program execution on RTL implementations, the functional simulator, and the ANN in the first three phases. However, these phases are only required once to generate the PMs. The time taken by the LLM in the first two phases was approximately 1102.46 seconds for SweRV, and 2060.69 seconds for RSD. Program execution on the SweRV took 67151.97 seconds, while execution on RSD required 102917.82 seconds. The program execution time on the Whisper functional simulator was significantly shorter, at 341.67 seconds. Training the PM using the ANN took 16.03 seconds for SweRV and 1737.27 seconds for RSD.

We compared our proposed methodology to the state-of-the-art performance estimation approach presented in [8]. That study evaluated four RTL implementations, achieving low errors for two of them. In contrast, SweRV and RSD showed higher errors, with MAPEs of 6.5% and 18.6%, respectively, indicating the need for further refinement. By focusing on SweRV and RSD, our methodology demonstrates significant improvements, reducing the MAPE by 61.54% for SweRV and 36.02% for RSD. These improvements provide higher precision in estimating executed clock cycles across multiple benchmarks. The enhancements are primarily due to the additional steps introduced in our work, such as improved data

extraction using LLM and conflict resolution strategies, which refine the handling of performance-related parameters from RTL documentation and source code.

V. Conclusion

In this paper, we proposed an LLM-assisted methodology for estimating the performance of embedded software on RISC-V processors. By combining the natural language processing capabilities of an LLM with the computational power of an ANN, we created PMs that offer both accuracy and efficiency. The LLM extracts performance-related parameters from RTL source code and documentation, while the ANN predicts performance using these parameters and dynamic instruction counts obtained from a functional simulator.

Our experiments with real-world benchmarks demonstrated highly accurate predictions, with MAPEs of 2.50% for SweRV and 11.90% for RSD. Compared to state-of-the-art approaches, our method reduced the MAPE by 61.54% for SweRV and 36.02% for RSD. These results indicate the potential of integrating LLM and ANN to enhance performance estimation workflows, particularly for hardware-software co-design in embedded systems requiring precise and efficient predictions.

Despite these promising results, several challenges remain. The LLM's limited context window required splitting source code into smaller chunks, which sometimes disrupted the logical structure. Future work will focus on advanced code-splitting techniques that preserve functionality and logical flow. Additionally, we aim to extend this methodology to more complex architectures and explore alternative ML models to further improve accuracy and scalability. We also plan to experiment with other LLMs and compare their results to assess their impact on performance estimation. These enhancements will refine the approach and expand its applicability to a broader range of systems.

References

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual. Volume I: Unprivileged ISA*, RISC-V Foundation, 2019.

[2] "SiFive," https://www.sifive.com/.

[3] "RISC-V Landscape," https://riscv.landscape2.io/?view-mode=grid.

[4] K. S. Kalyan, "A survey of gpt-3 family large language models including chatgpt and gpt-4," *Natural Language Processing Journal*, p. 100048, 2023.

[5] J. Y. Khan and G. Uddin, "Automatic code documentation generation using gpt-3," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–6.

[6] J. Ahn, R. Verma, R. Lou, D. Liu, R. Zhang, and W. Yin, "Large language models for mathematical reasoning: Progresses and challenges," *arXiv preprint arXiv:2402.00157*, 2024.

[7] M. S. Oyamada, F. Zschornack, and F. R. Wagner, "Applying neural networks to performance estimation of embedded software," *Journal of Systems Architecture*, vol. 54, no. 1-2, pp. 224–240, 2008.

[8] W. Zhang, M. Goli, M. Hassan, and R. Drechsler, "Efficient ML-based performance estimation approach across different microarchitectures for RISC-V processors," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 693–699.

[9] W. Digital, "SweRV RISC-V Core," https://github.com/chipsalliance/Cores-VeeR-EH1/tree/1.0.

[10] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue *et al.*, "An open source fpga-optimized out-of-order RISC-V soft processor," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 63–71.

[11] CHIPS Alliance, "Whisper," https://github.com/chipsalliance/VeeR-ISS.

[12] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "Taclebench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[13] VERIPOOL, "Verilator," https://www.veripool.org/verilator/.

[14] T. Ta, L. Cheng, and C. Batten, "Simulating multi-core RISC-V systems in gem5," in *Workshop on Computer Architecture Research with RISC-V*, 2018.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[16] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020.

[17] M. Montón, "A RISC-V systemc-tlm simulator," *arXiv preprint arXiv:2010.10119*, 2020.

[18] C. Marantos, K. Salapas, L. Papadopoulos, and D. Soudris, "A flexible tool for estimating applications performance and energy consumption through static analysis," *SN Computer Science*, vol. 2, pp. 1–11, 2021.

[19] J. Silveira, L. Castro, V. Araújo, R. Zeli, D. Lazari, M. Guedes, R. Azevedo, and L. Wanner, "Prof5: A RISC-V profiler tool," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2022, pp. 201–210.

[20] "Spike RISC-V ISA simulator," https://github.com/riscv-software-src/riscv-isa-sim.

[21] M. Ali, E. Aliagha, M. Elnashar, and D. Göhringer, "RV-ProViler: Evaluating RISC-V ISA for application-specific requirements," in *2024 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2024, pp. 1–7.

[22] "RISC-V GNU Compiler Toolchain," https://github.com/riscv-collab/riscv-gnu-toolchain.

[23] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert, "Early execution time-estimation through automatically generated timing models," *Real-Time Systems*, vol. 52, pp. 731–760, 2016.

[24] W. Zhang, M. Goli, and R. Drechsler, "Early performance estimation of embedded software on RISC-V processor using linear regression," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2022, pp. 20–25.

[25] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*. PMLR, 2019, pp. 4505–4515.

[26] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," in *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2023, pp. 1–6.

[27] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "LLM-guided formal verification coupled with mutation testing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–2.

[28] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–31, 2024.

[29] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "LLM for SoC security: A paradigm shift," *IEEE Access*, 2024.

[30] "Chroma," https://js.langchain.com/v0.1/docs/integrations/vectorstores/chroma/.

[31] "GPT-3.5," https://platform.openai.com/docs/models/gpt-3-5-turbo.

[32] "LangChain," https://www.langchain.com/.

[33] "CharacterTextSplitter," https://js.langchain.com/v0.1/docs/modules/data_connection/document_transformers/character_text_splitter/.

[34] SBERT, "all-MiniLM-L6-v2," https://sbert.net/.

[35] "Vector store-backed retriever," https://python.langchain.com/v0.1/docs/modules/data_connection/retrievers/vectorstore/.

[36] "FlashRank reranker," https://python.langchain.com/v0.1/docs/integrations/retrievers/flashrank-reranker/.

[37] "TensorFlow," https://www.tensorflow.org/.

[38] Keras, "Random search," https://keras.io/api/keras_tuner/tuners/random/.

[39] "ChatPromptTemplate," https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html.

[40] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," *arXiv preprint arXiv:1802.03426*, 2018.