SpeedGen: Enhancing Code Efficiency through Large Language Model-Based Performance Optimization

Nils Purschke*
Technical University of Munich
Munich, Germany
nils.purschke@tum.de

Sven Kirchner*
Technical University of Munich
Munich, Germany
sven.kirchner@tum.de

Alois Knoll
Technical University of Munich
Munich, Germany
k@tum.de

***Abstract*—We present SpeedGen, a novel framework that uses Large Language Models (LLMs) to automate code performance optimization. SpeedGen is designed to address software performance bottlenecks using a feedback-driven approach that profiles code to identify inefficiencies and iteratively refines the code to improve execution speed.**

We conducted a comprehensive evaluation of SpeedGen's capabilities across diverse codebases, benchmarking its performance against a leading large language model. Our results show that SpeedGen consistently reduces execution time and delivers significant performance improvements in various scenarios. The framework's ability to adapt to different domains underscores its scalability and robustness, making it a valuable tool for optimizing code in a wide range of applications.

A key strength of SpeedGen is its ability to maintain the functional correctness of the code while achieving significant performance gains. This feature ensures that the optimized code remains reliable even when it undergoes significant transformations. By automating the optimization process, SpeedGen minimizes the need for manual intervention, streamlining the software development lifecycle and reducing time-consuming performance tuning efforts.

The introduction of SpeedGen marks a major step forward in the integration of LLMs into software engineering and paves the way for future research and development in this area. With its ability to improve performance without compromising code integrity it lays the foundation for more advanced automated code optimization techniques, simplifying software development.

***Index Terms*—Automatic Programming, Language Models, Performance, Reliability**

I. Introduction

Large Language Models are becoming increasingly important in software engineering [1]. Given a natural language instruction, they are capable of writing code in any programming language in which they have been trained. This ability allows developers to streamline their workflows, automate repetitive tasks and prototype new features with unprecedented speed. As a result, LLMs are changing the landscape of programming, making it more accessible and enabling developments that were previously out of reach [2].

The quality of code generated by LLMs is highly dependent on the model architecture and the underlying training data [3]. It has long been a challenge to beat human performance at coding [4]. While the current generation of LLMs, such as GPT-4 [5], can perform basic coding tasks with high accuracy, [6] shows that they have not yet reached the standard of human software development. There is still a need for extensive review of all code written by LLMs, and they are not yet able to reliably understand complex coding tasks [7].

A critical aspect of software development is optimizing code performance. This task is time-consuming and requires extensive knowledge of code performance profiling and various optimization strategies [8]. Despite its complexity, it is essential in many fields, ranging from low-level and high-performance computing to the front-end development of mobile applications [9]. Rather than manually optimizing code, a language model can be used, although this still requires considerable manual effort. The process start with generating a prompt and using a language model to optimize the corresponding code. The supposedly optimized code is then reintegrated into the code base. After that, it is critical to verify that the supposedly optimized code maintains its correctness and that its performance has indeed be improved compared to the original implementation. If the optimization proves inadequate or results in incorrect functionality, the process must be repeated.

In this work, we provide a pipeline to fully automate the process of improving the runtime performance of programming code. By doing so, we aim to simplify future software development and minimise error-prone and time-consuming tasks that require a good understanding of code optimization techniques.

In more detail, the following main contributions are presented in this paper:

- We propose a new fully automated pipeline for code performance improvement, which validates that the generated code is indeed faster.
- We introduce a robust error handling mechanism within our pipeline, which significantly reduces the rate of failed

* Equal contribution

This research was funded by the Federal Ministry of Education and Research of Germany (BMBF) as part of the CeCaS project, FKZ: 16ME0800K.

optimizations due to code generation errors.

- We compare the performance of our pipeline with a state-of-the-art large-language model, demonstrating its superior effectiveness in optimization.

II. Related Work

The application of large language models to software engineering tasks is an active area of research, with a multitude of recent advances but also ongoing challenges. Previous studies have highlighted both the potential and limitations of LLMs for code generation, addressing issues of accuracy, efficiency and applicability across different programming languages and domains.

A. Large Language Models and Code Generation

As Natural Language Processing (NLP) technologies have advanced, more complex tasks have become feasible through the introduction of large language models such as GPT-3 [10]. They are the state-of-the-art basis for automated code generation [11]. With significant improvements in recent months, automated code generation is closer than ever. Over the last two years, dozens of models have been released specifically for the task of automated code generation, such as WizardCoder [12], CodeBERT [13] based on BERT [14], CodeGen [15] or Meta's Code LLama [16], many of which are open source.

Apart from that, large language models can be used for many different related tasks or subcategories, such as automated program repair [17], automated unit test generation [18], automated coding comments generation [19] and assisting the programmer with coding suggestions [20]. Consequently, there are also models specifically trained to understand and generate code with respect to flexibility and performance [21], [22]. [23] provides a systematic overview of the many ways in which machine learning is used for code generation, reviewing 37 studies on machine learning-based code generation and classifying them into categories such as description-to-code conversion.

There are different LLMs specifically designed for code related tasks, like CodeT5Mix a versatile model combining encode-decoder Transformers that is pre-trained on divers tasks across nine programming languages and can be adapted flexibly to various tasks during fine tuning [21] or Codex a GPT language model fine-tuned on GitHub code, demonstrating superior Python code-writing capabilities over GPT-3 and GPT-J [24]. Also notable is CodeLLama a family of large language models based on Llama 2, which offers state-of-the-art performance for code-related tasks with various specializations and parameter sizes, supporting large input contexts and infilling capabilities [16] .

B. Code Correctness and Performance

When writing software, the correctness and performance of the code are particularly relevant quality characteristics. While logical correctness can be checked by unit testing, further effort is required to validate the performance of the code.

Unit testing focuses on verifying that individual components or small groups of related components within software work as intended [25]. This approach helps to identify and fix bugs early in the development cycle, resulting in more reliable and maintainable code. By independently validating the functionality of each unit, developers can achieve higher code quality and facilitate code refactoring.

In comparison to software correctness, performance is highly dependent on the underlying hardware, as dedicated hardware can significantly speed up processing tasks, and factors such as clock frequency and core isolation directly affect how efficiently software runs [26]. In addition, effective resource allocation through advanced scheduling and caching mechanisms can mitigate bottlenecks and further improve performance [27]. When rewriting code to improve performance, it is important to understand the influence of these factors and to eliminate related measurement inaccuracies in order to focus on the relevant parameters. Therefore, the following factors need to be considered when evaluating code performance on hardware:

1) Allocation and Mapping: Hardware allocation for software involves distributing software tasks across different hardware resources, such as processors, memory, and storage devices, to optimize the use of these resources. This results in differences in code performance because efficient allocation minimises bottlenecks and ensures that tasks are executed on the most appropriate hardware, resulting in faster and more efficient software operation [28].

2) Scheduling: Scheduling in software involves prioritising and ordering tasks for execution on available hardware resources to maximise efficiency and minimise idle time. Scheduling determines the exact time instance at which a task is executed. It is classified on the basis of its constraints, such as time and resources [29]. Effective scheduling has a significant impact on code performance by ensuring optimal resource utilisation, reducing task wait times, and improving overall system throughput.

3) Caching: As the number of central processing unit (CPU) cores is increased, cache memory is one of the key factors regarding the performance and power of the CPU [30]. Caching is a technique to reduce peak traffic rates by prefetching popular content into memories at the end users [31]. It involves storing frequently accessed data in faster storage locations, close to the CPU, such as L1, L2, and L3 caches in order to reduce retrieval times and improve performance. Effective caching can greatly enhance code performance by minimizing latency and reducing the load on slower, primary storage, leading to faster data access and overall system efficiency.

4) Dependencies: In addition to the hardware dependencies, the software performance is heavily influenced by the choices made in its design and implementation. Choosing the right algorithms and data structures is critical, as these can drastically affect execution speed [32]. The choice of libraries also plays an important role; some libraries may introduce unnecessary overhead or may not be optimized for performance, so selecting efficient, well-maintained libraries is essential [33], [34]. Software design methodologies, such as modular design and object-oriented programming, promote maintainability but can introduce performance trade-offs due to additional layers of abstraction [35]. Minimising the use of blocking operations and optimizing for asynchronous processing can help reduce execution delays [36]. Regular use of profiling tools allows developers to identify and optimize slow code paths. Finally, balancing clean, modular design with performance considerations ensures that the software is both maintainable and fast.

C. Performance Optimization

Automated code optimization techniques have been a focus of research for many years. These techniques range from code refactoring to complex compiler optimizations. For example, optimizations such as loop unrolling, inlining and dead code elimination are traditional methods used to improve performance [37].

[38] uses information about the underlying processor architecture and loop kernel characteristics to find the optimal optimization transformation to improve the code. Another method proposed by [39] introduced code perforation as a technique for improving the performance of computations by selectively skipping or perforating parts of the code that have minimal impact on the final output. This allows a trade-off between accuracy and performance, where a small amount of accuracy can be sacrificed to achieve significant performance improvements.

Recent approaches focus on using advanced AI-based tools to improve code speed. The authors of [40] present AI-powered compiler techniques that optimize deep learning code for CPUs by applying high-level optimizations for cache utilisation and low-level optimizations for effective single instruction, multiple data (SIMD) vectorization, achieving significant performance improvements over baseline and existing deep learning compilers. CompilerGym [41] advances the field by providing a comprehensive and scalable platform that integrates multiple compiler problems, optimization targets, and offline datasets, surpassing previous tools such as OpenTuner [42] and YaCoS [43] in terms of flexibility and ease of use for both reinforcement learning and autotuning research.

DeepPERF author's pre-trained BART [44], a transformer based language model with less then one billion parameters, on code completion and fine-tuned it on code performance improvements [45]. During their experiments they showcased that language models can indeed improve code performance. Additionally, they identified the code integrity as a critical thread, due to the adaption of the code, potentially introducing bugs. Using modern LLMs like GPT 3.5 [46] to optimize the code's runtime performance leads to significant improvements, surpassing human optimizations at times. This was shown by the authors of [47], which also evaluated different prompting techniques and fine-tuning of LLMs for performance optimizations.

D. LLM-Benchmarking

Specific benchmarks are used to compare the performance of LLMs. [48] presented the HumanEval+ dataset, which is based on the HumanEval dataset [24]. Showing current large language models achieving success rates of up to 86% on general coding problems. [49] introduced a benchmarking tool that evaluates LLMs based on both code correctness and performance metrics.

Language models themselves can also be used to benchmark other models: PandaLM is a judge model for automatic evaluation and hyper-parameter optimization in LLM instruction tuning, which provides a robust alternative to traditional methods by focusing on subjective factors such as clarity, conciseness and instruction compliance. This approach not only improves model performance, but also increases the reproducibility and efficiency of evaluation processes [50].

LLMs can suffer from the inclusion of test data in their training data, thereby distorting the benchmark evaluation. LiveBench addresses the shortcomings of existing LLM benchmarks by introducing a frequently updated, contamination-free evaluation framework based on objective ground-truth scoring across diverse and challenging tasks, unlike benchmarks that suffer from test-set leakage or bias in LLM and human-based judgments [51].

E. Future Directions and Applications

The future of LLMs in software development looks promising, with ongoing research aimed at overcoming current limitations. The potential for LLMs to support debugging, software design and other areas of software development is being explored [52], [53]. Studies suggest that with further advances in model architectures and training techniques, LLMs could play a more integral role in software development workflows [54].

However, an approach that uses LLMs to generate performance-optimized code in a reliable, feedback-based manner is still lacking [55]. This paper builds on the foundations of current research by proposing a novel method to do so. Through these contributions, we aim to advance the field towards more reliable and efficient LLM-based software development.

III. SpeedGen

To address the challenges of manual code performance optimization, we developed SpeedGen, an automated pipeline that leverages large language models to streamline the optimization process. SpeedGen employs various mechanisms and tools to ensure the optimized code maintains correctness while achieving higher performance than the original code. Implemented in Python, SpeedGen is capable of optimizing Python code and designed for easy extension to support other programming languages. In case SpeedGen is unable to enhance performance, it explicitly communicates this outcome, setting it apart from approaches that rely solely on LLMs. It consists of the following pipeline phases (Fig. 1):

- Preprocessing

- Performance Evaluation
- Code Generation
- Decoding
- Integrity Verification

A. Preprocessing

The first phase of SpeedGen involves installing all necessary dependencies required for the original code to run. SpeedGen automatically detects and installs these dependencies by parsing them from the code. This step ensures that the environment is correctly set up with the appropriate libraries and potential errors related to missing libraries are avoided.

The phase is triggered again after the seemingly optimized code has been written by the LLM to ensure that all dependencies of the optimized code are also installed. This reinstallation process is critical because new dependencies, or updated versions of existing dependencies, may be required for the optimized code to work correctly. This automatic dependency management reduces setup time and eliminates human error in configuring the environment.

B. Performance Evaluation

An important aspect of pre-optimization is the identification of code bottlenecks that consume a high proportion of processing time. This allows for an efficient optimization process where the identified most critical areas of the code can be targeted. By default, LLMs are not equipped with profiling tools to do this. Therefore, providing an LLM with information about these slow code segments is expected to increase its effectiveness in reducing code runtime.

At the same time, profiling provides information about the overall execution speed of the code, acting as a benchmark. It allows comparisons between the original and the optimized code, ensuring that the optimized version is actually faster.

SpeedGen uses profiling for both of those aspects. It measures the performance of code under different input scenarios using cProfile [56], a deterministic Python profiler. In detail cProfile works by instrumenting function calls as the program is executed, allowing it to gather precise timing information without relying on sampling, ensuring that all function calls are accurately captured. The analysis provides detailed statistics about the program execution, such as total execution time, the number of calls to each function and the time spent per function call, providing a comprehensive understanding of how different parts of the code contribute to the overall runtime.

Moreover, SpeedGen can easily be extended to support additional benchmarks, providing flexibility in performance evaluation.

In the pipeline, the performance evaluation step is first applied when the performance of the original code is measured to establish a baseline. The runtime of any apparently optimized code written by the LLM is then later compared to this baseline performance to qualify it as optimized.

In addition, the profiling results serve as auxiliary information for the LLM to improve the optimization process. Initially, the baseline profiling result of the original code is used as input. During optimization, each profiling result of the apparently optimized code generated by the LLM is used as feedback input for the next prompt in case the optimization was unsuccessful. Performance optimization is considered successful if the new code outperforms the old code, with a performance difference greater than zero. To generalize the results and reduce noise, this performance measurement is performed ten times. Finally, the average of these measurements is taken. This iterative feedback mechanism ensures that the LLM is continuously guided towards producing more efficient code, refining its output based on concrete performance data.

Before profiling the code, it is critical to establish consistent initial conditions for each measurement to ensure the accuracy and reliability of the performance data. One of the most important steps in achieving this is to flush the CPU cache before each run. This action clears any residual data from previous calculations, thus eliminating potential cache hits that could skew the profiling results. By resetting the cache state, we ensure that each execution starts with an identical cache configuration, providing a fair and controlled environment for performance evaluation. In addition, to minimize the impact of context switching - which can occur when the operating system interrupts the current process to execute another - processes are given real-time priority where supported. Real-time priority helps maintain the continuity of process execution by reducing interruptions from other processes, resulting in more consistent profiling results. In addition, to improve isolation and reduce interference from other tasks, each process is bound to a specific CPU core using core affinity settings. By restricting a process to a specific core, we can prevent the operating system from moving the process between cores, which could otherwise introduce variability due to differences in core performance or cache state as well as the time it takes to move the context from one to another core.

These measures - cache flushing, real-time priority scheduling and core affinity restrictions - together contribute to a controlled and repeatable profiling environment, helping to ensure that performance measurements accurately reflect the behaviour of the code under consistent and reproducible conditions.

C. Code Generation

Another key component of the pipeline is the LLM. Its inference is programmed so that new LLMs can be integrated by adding just a few lines of code. The LLM is responsible for optimizing the code, using information from the other pipeline stages such as performance profiling and correctness checking.

The first prompt, the system prompt, contains a detailed guide that instructs the LLM on optimizations relating to data structure, algorithms, parallelization, redundancy and more. This predefined input is used to configure and control the behaviour of the language model. In the first guidance phase, the original code and the generated baseline profile are fed into the LLM, together with a request asking for performance optimizations while maintaining the original functionality.

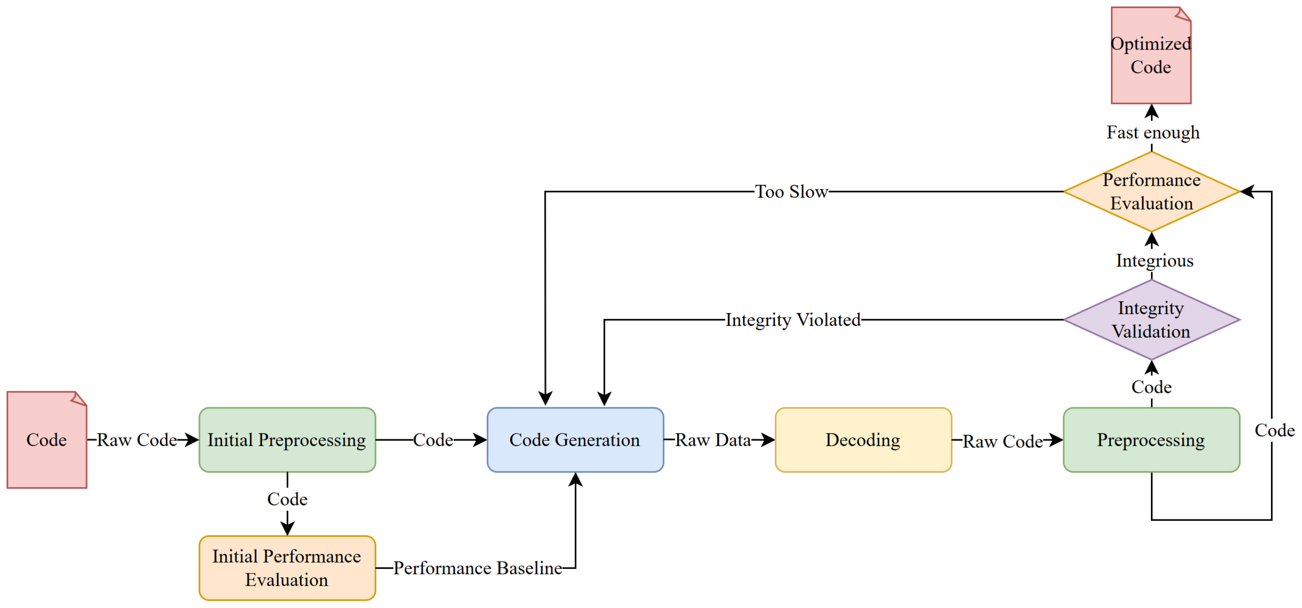

Fig. 1: Pipeline Overview: The SpeedGen pipeline is organized into various stages, each representing a specific tool or process, such as code generation or performance evaluation. Some of these stages are utilized multiple times throughout the pipeline. The progression to the next stage may vary based on the output of the current stage. The end result is either optimized code or a failure if the code generation loop has been repeated a specified amount of times.

If a tool, such as the performance evaluation tool, detects a mismatch, its feedback is used to generate a new, customized prompt tailored to that specific tool. This prompt, along with the feedback, is sent to the LLM, which then attempts to correct the error by rewriting the code. If errors persist after a configurable number of retries, the current session is terminated and the code optimization process for that session is marked as failed. A new empty session can then be initiated to retry the optimization. This is useful if the LLM has reached a dead end. When a configurable number of restarts is reached, the entire optimization for the current problem is marked as failed, preventing SpeedGen from getting stuck in an endless loop if the optimization is not possible.

Another aspect of prompting is the token and memory limit. During the prompting process, the system continuously monitors the token count of both the input and the output. If the token count approaches the model limit or the memory limit, the input is dynamically adjusted within the acceptable range. This involves heuristically truncating less critical parts of the code or feedback, while ensuring that the essential parts required for optimization are retained.

D. *Decoding*

After receiving a response from the LLM, SpeedGen decodes the suggested optimizations and translates them into executable code. This phase involves parsing the LLM output, handling code formatting and integrating the proposed changes into the testbed. The decoded response must be syntactically correct. If no answer is found, the LLM is reminded to write optimized code and to use the Markdown [57] code block formatting to mark the optimized code, a format commonly used by LLMs and therefor used by us to identify the code location in the answer. This step ensures that the output of the LLM is in a usable state, ready for testing and further validation.

E. *Integrity Verification*

Once the optimized code has been generated, SpeedGen performs an integrity check to ensure that the modifications have not changed the intended functionality. This involves running a series of tests and comparing the results generated by the code with the expected results. Any discrepancies are flagged and the optimization is discarded if correctness is compromised. In addition, the LLM receives feedback on the failed test cases and is prompted to fix these problems. This iterative process ensures that the final output is not only faster, but also retains the original functionality.

IV. Evaluation

In the following, we evaluate SpeedGen and test its ability to optimize code performance while ensuring correctness. We evaluate SpeedGen on a large dataset, namely BigCodeBench, using a blank virtual Python environment [58]. Performance metrics were collected using standardized benchmarks to ensure consistency and reliability of results.

A. *Hardware Setup*

To evaluate the performance of the code, we use a high performance computer (HPC) equipped with a powerful graph-

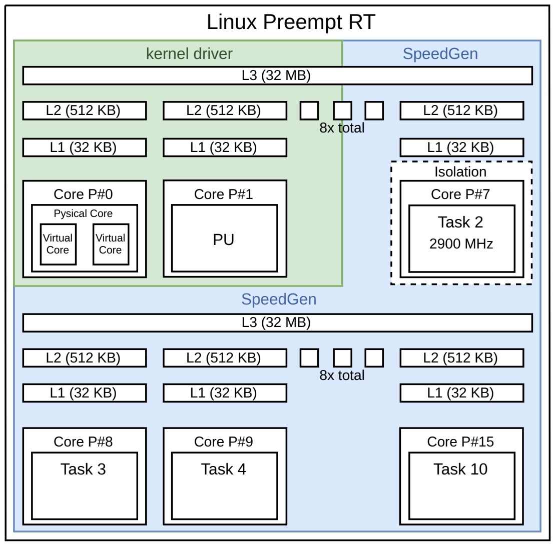

Fig. 2: CPU Architecture of the HPC shown in Table I, consisting of a total of 16 physical cores. SpeedGen is exclusively assigned 10 of those cores, while the operating system uses the remaining 6.

ics processing unit (GPU), as detailed in (Table I). The HPC setup with a GPU is essential for running our LLMs locally, as it provides the necessary computing power and memory to handle large model queries. This configuration also includes parallel processing capabilities through the server-grade CPU, which significantly reduces execution times and therefore increases efficiency.

Name	Generation and Evaluation HPC
CPU	AMD TRP 5955WX - 16 Cores at 4.0 GHz
GPU	RTX 4090 - 24 GB VRAM
RAM	64 GB DDR4-3200 RAM

TABLE I: Hardware setup - The SpeedGen Generation and Evaluation Computer features a powerful graphic processor with a lot of video random access memory (VRAM) and a high performance CPU, both optimized for increased processing power and parallel computing.

B. Hardware & Software Mapping and Allocation

When evaluating the performance of software, a consistent test environment is important to achieve reproducible results. External factors that affect software performance should be eliminated. Following this advice, we created an isolated environment.

First, we apply the PREEMPT_RT patch to our Linux kernel [59]. This patch allows the kernel to interrupt a running process to switch to a higher priority process, improving system responsiveness, especially for real time applications, thus promoting the operating system to be real time capable. SpeedGen takes advantage of this feature. It ensures that critical processes undergoing performance measurements receive immediate CPU attention, reducing latency and avoiding interruptions to other processes by giving performance measurement processes real-time scheduling priority.

Having created a closed environment on the software side, we need to make sure that the computations on the hardware side are always done in the same way. We do this through several measures. First, we disable simultaneous multithreading, which allows a single physical CPU core to handle multiple threads simultaneously, potentially improving multitasking performance. However, when performing a performance evaluation, it is essential to disable the logical cores that do the parallel work. This ensures that the results reflect the true processing power of each core without the influence of virtual parallelism using a single physical core. Therefore, instead of a total of 32 cores, 16 physical cores are available and can be used in our system, as shown in Fig. 2.

Isolating CPU cores 6 to 15, dedicates them exclusively to the performance tests, ensuring that no other processes interfere with the measurements. Cores number 0 to 5 are reserved for the operating system. This isolation provides more accurate and consistent performance results by minimising the impact of operating system activity and background tasks on the test environment.

To further improve measurement reproducibility, the clock frequency of each core is set to a fixed frequency of 2900 MHz, eliminating variability due to dynamic frequency scaling. CPU frequency boost is also disabled for all cores. As the CPU cache and exact task allocation is handled by SpeedGen, the evaluations can be run with a high consistency.

C. LLM and Dataset

We use Llama 3.1-8B Instruct as our large language model, the latest LLM from Meta, for several compelling reasons [60]. Llama 3.1-8B Instruct can be deployed locally, ensuring data security and control over our computing resources. It is also open source, which allows for extensive customisation and transparency. Llama 3.1 is designed with advanced skills in general knowledge, tool use and programming, making it effective in optimizing code performance while ensuring correctness.

We use BigCodeBench as a baseline for evaluating our dataset because of its extensive range of libraries and diverse programming tasks, as well as the included unit test to validate the correctness of the code. Although it is a public dataset, we mitigate data leakage by ensuring it could not have been part of LLama3's training data due to its release date and the absence of tasks related to code optimization or appropriate solutions. In addition, the use of BigCodeBench allows us to benchmark our results against a recognised standard in the research community. Since it is based on real-world datasets and widely used for benchmarking, it provides a realistic baseline for codes performance. It also ensures that SpeedGen can be evaluated in a variety of real-world scenarios, providing a robust assessment of its capabilities when used productively in real-world software development.

By using SpeedGen, we evaluate and improve the code performance of the BigCodeBench dataset, so that the optimized results could potentially be used for a new performance-optimized code dataset.

V. Results

We started by evaluating the performance of the original code in the BigCodeBench dataset. This provided us with baseline performance metrics. We then used SpeedGen to optimize the code, employing these initial performance metrics to gauge improvements. By comparing the optimized code created by SpeedGen with the original baseline, we systematically measured performance enhancements, validating the effectiveness of SpeedGen's optimization processes. This approach ensures a robust and thorough evaluation of SpeedGen's capabilities.

A. Exemplary Process

The process flow of the SpeedGen pipeline becomes clearer with the help of an example (Fig. 3). As an illustration the entry number zero of the BigCodeBench dataset is used as the baseline code, which should be improved by SpeedGen (Fig. 3a).

First, following the pipeline flow, the initial preprocessing was handled. The libraries used in the code were parsed from it and if there were missing, non-system libraries found, they would have been installed. In this case only system libraries were used, so this step lead to no changes.

After the dependency installation, the baseline performance evaluation was performed using cProfile, resulting in profiling statistics. This includes, among other metrics, how much time is spent on the functions called in the code and a total summed up runtime performance of around 40.9 seconds. The generated profiling statistics and a detailed guideline on improving the given code were provided as input for the first request to the LLM. These inputs included suggestions such as using parallelism or leveraging libraries, along with the baseline code, to enhance performance. Subsequently the LLM generated a response from which the code was successfully extracted. The dependency installation was skipped, since there were no dependencies used at all. It was then checked for correctness using the unit tests provided by the BigCodeBench dataset for this entry. The correctness check failed, due to the LLM replacing the original code with a call to the than nonexistent method included in the original code and printing the result. It was consequently notified of its mistake, by including the results of the failed unit tests as well as a request to fix its errors, in the next message to it.

This request failed again, due to the LLM now writing two functions to replace the original code. One was the unmodified original function, and the other one was the seemingly optimized function. Since the name of the functions both differ from the original function name all unit tests failed. The third try failed as well, with no relevant changes made by the LLM. This lead to a restart, meaning a clean start of the conversation, because we assume that the LLM has navigated itself into a dead end after three unsuccessful tries.

This is demonstrated here empirically by the practically unchanged code produced by the LLM comparing the second and the third try.

On the start of the next try the LLM generated correct code. Consequently the unit tests passed, and the next pipeline stage was reached, the profiling of the seemingly optimized code. It determined, that the code is slower than the baseline. Following this, the LLM was informed of its failure and equipped with the profiling statistics of its code, to fix the performance bottlenecks. The second try failed as well with no significant changes compared to the previous try. Its was therefor asked to improve the code performance, once more.

This once again resulted in code being generated by the LLM, which was successfully decoded form the response (Fig. 3b). Once again, all dependencies of the new code were installed. In this case, leading to a change: The installation of the numpy library, which is based on highly optimized C code [61], [62]. The resulting, allegedly correct and performance optimized code, was again checked for correctness, which succeeds. After profiling, it is determined, that the code is in fact more than twice as fast the original code. Therefore the pipeline finishes successfully.

B. Baseline Performance

For the performance evaluation using the BigCodeBench dataset a preprocessing step is run first. It has the goal of identifying bottlenecks and time-consuming functions as well as library usage and ensures that the performance measurements are reproducible.

This involves analyzing which parts of the code generation process take the longest, examining the efficiency of various functions and libraries used, and assessing their impact on the overall performance. The evaluation aims to pinpoint specific areas where optimization can significantly reduce execution time and improve the performance of LLMs in handling complex programming tasks.

When calculating the performance statistics we perform the performance measurement 10 times and then calculate the average as well as the standard deviation of the results. We filter the code for those who leave significant room for improvement and a low relative standard deviation.

C. Overall Correctness

To quantify the effectiveness of the code generation process, we conducted a thorough analysis of the correctness of the generated code. Out of the total number of code examples in the dataset, the number of instances where the generated code was correct (meaning it produced the expected output) was recorded. Specifically, the large language model was able to generate correct code for 799 out of 1077 examples.

The results indicate that the large language model demonstrated a substantial ability to generate correct code, achieving a code correctness rate of 74% across the provided dataset.

```python
import itertools
from random import shuffle
def task_func(numbers=list(range(1, 3))):
    permutations = list(itertools.permutations(numbers))
    sum_diffs = 0

    for perm in permutations:
        perm = list(perm)
        shuffle(perm)
        diffs = [abs(perm[i] - perm[i+1]) for i in range(len(perm)-1)]
        sum_diffs += sum(diffs)

    avg_sum_diffs = sum_diffs / len(permutations)

    return avg_sum_diffs
```

(a) Baseline code as provided through the BigCodeBench dataset

```python
import itertools
import numpy as np

def task_func(numbers=list(range(1, 3))):
    """
    Calculate the average sum of absolute differences between adjacent elements
    in a randomly shuffled permutation of the input numbers.

    Args:
        numbers (list): A list of numbers to generate permutations from.

    Returns:
        float: The average sum of absolute differences.
    """
    # Generate permutations
    perms = list(itertools.permutations(numbers))

    # Use NumPy for efficient array operations
    np_perms = np.array(perms)

    # Shuffle each permutation and calculate the sum of differences
    diffs = np.sum(np.abs(np.diff(np.array([np.random.permutation(perm) for perm in np_perms]))), axis=1)

    # Calculate average sum of differences
    avg_sum_diffs = np.mean(diffs)

    return avg_sum_diffs
```

(b) Performance optimized code generated by SpeedGen

Fig. 3: Using the baseline code (Fig. 3a) taken from the BigCodeBench dataset, SpeedGen is able to iteratively improve the performance, while ensuring code correctness, resulting in performance optimized code (Fig. 3b).

This finding underscores the potential of large language models in automating code generation tasks and enhancing code performance. Since still in very active development future refinement of the capabilities of large language models will lead to further improvements in terms of the correctness of the generated code.

D. Performance Improvement

In addition to evaluating the correctness of the generated code, a key objective of this study was to assess the performance improvements achieved through the use of our code generation tool, SpeedGen.

Table II presents the evaluation of SpeedGen on the BigCodeBench dataset, comparing the baseline performance to the performance after applying SpeedGen across various tasks. The ”Total Performance Improvement” column shows the difference in execution time (in milliseconds), and the ”Result” column indicates whether SpeedGen was able to create the code with an improved or slower performance or could not pass the unit tests (”Incorrect”). The ”Speedup Factor” quantifies the relative improvement as a percentage. It was calculated as follows:

Task ID	Baseline	SpeedGen	Total Performance Improvement [ms]	Result	Speedup Factor [%]
	Performance [ms]	Performance [ms]			
0	40886	14519	26367	Faster	181.60
1	3	5	-2	Slower	-40
2	7	6	1	Faster	16.67
3	6	4	2	Faster	50
4	5	6	-1	Slower	-16.67
5	7	6	1	Faster	0
...	...	...	...	...	
579	273	159	114	Faster	71.7
...	...	...	...	...	
1074	14	10	4	Faster	40
1075	11	10	1	Faster	10
1076	4	-	-	Incorrect	-
1077	44	-	-	Incorrect	-
Analysis	-	-	-	450 Faster 349 Slower 278 Incorrect	Average Speedup: **101.0**

TABLE II: Evaluation results of SpeedGen on the BigCodeBench dataset. SpeedGen increases the code performance by an average of 101,0% compared to the baseline.

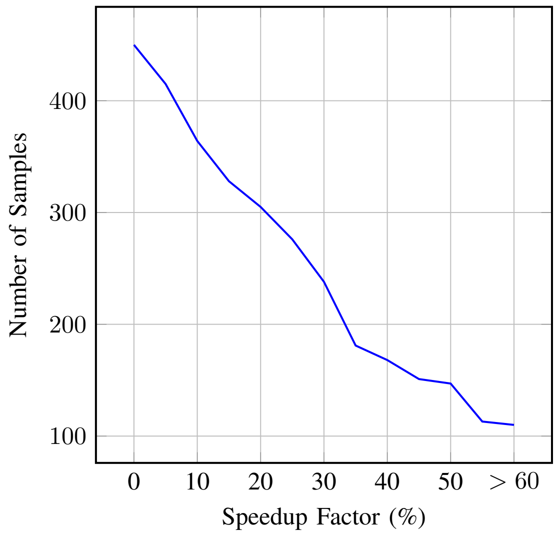

Fig. 4: Code performance improvements achieved by SpeedGen on the BigCodeBench dataset.

$$SpeedupFactor = \frac{Runtime_{Baseline}[ms]}{Runtime_{SpeedGen}[ms]} * 100\% - 100\% \quad (1)$$

From the results generated by profiling the code programmed by the LLM based on the BigCodeBench dataset and comparing the performance we get the following results: From the 1077 used samples SpeedGen was able to transform the code 799 times without violating its correctness. The generated code exhibited a marked reduction in execution time across the majority of the tasks. Through all of the 799 correct code creations, SpeedGen was able to achieve an average Speedup Factor of 101% compared to the original code, increasing the performance in 466 cases while being slower in 333 of them. Important to note is that SpeedGen itself identifies both code inaccuracies and performance inefficiencies. Consequently, any generated code that is either incorrect or fails to outperform the existing baseline would naturally not be used by the end user. This distinguishes SpeedGen from existing solutions in this area, generating potentially faster code but not automatically validating the performance gains and the correctness of it.

Fig. 4 depicts the distribution of the Speedup Factor achieved by SpeedGen across the BigCodeBench dataset. As the Speedup Factor increases, the number of samples exhibiting that level of improvement declines. The curve exhibits a predominantly linear trend, but with features characteristic of an inverse exponential decay, indicating a diminishing rate of performance improvements at higher levels. This suggests that as performance increases, the frequency of substantial gains decreases, consistent with a decelerating improvement trend often observed in such systems. This indicates that half of the samples performance improvements appear in the area smaller than 20%, while 110 samples achieve performance gains higher than 60%.

E. Detailed Analysis

While our results in Table II show significant improvement in performance when using SpeedGen for code performance optimization we want to further state the quality of the results. Therefore we post processed the dataset by filtering high quality code examples with a focus on reproducibility. The baseline, defined by the BigCodeBench dataset was analyzed regarding performance. We conducted the performance analysis 10 times for each code snippet and calculated the average performance as in Table II. We then filtered all the tasks that have a runtime lower than 10 milliseconds. Therefor we only keep the results with a significant runtime that allows a high quality performance analysis. Besides all efforts a small deviation in the performance between the 10 runs can still be noticed. We filtered the dataset again only keeping the samples that have a lower standard deviation in runtime than 0.075. As a result the dataset shown in Table III is generated.

SpeedGen Evaluation Dataset (runtime & deviation)	
Derived from	BigCodeBench
Performance runtime	< 10 ms
Standard deviation	< 0.075
Number of Entries	146

TABLE III: Performance-Baseline Dataset filtered by performance runtime and standard deviation.

To emphasise the added value of SpeedGen and to demonstrate its superiority over purely large language model-based performance optimization, we conducted a comparative study using Llama 3.1. The experimental conditions were controlled to ensure consistency, mirroring those used for SpeedGen. This included identical basic prompting and equivalent hardware and software constraints. Additionally, Llama 3.1 was tasked with optimizing code performance for each task under identical conditions, but without incorporating the SpeedGen adaptations. Each task allowed the model three fresh attempts to achieve optimization. When giving both approaches the task to optimize the performance of the given dataset from Table III the results as shown in Table IV emerge. SpeedGen shows a clear advantage over Llama 3.1 by providing more consistent and reliable performance optimization across different tasks. It successfully generated optimized code for 67 tasks compared to 53 in Llama 3.1, demonstrating its ability to handle a wider range of scenarios. In addition, SpeedGen achieved a higher average speedup factor of 285.19%, significantly better than Llama 3.1's average of 204.63%. This indicates that SpeedGen not only optimizes more tasks, but also delivers greater performance improvements on average, making it the superior choice for consistent and effective optimization.

VI. Conclusion

In this paper, we presented SpeedGen, an automated framework that uses large language models to effectively optimize code performance. SpeedGen systematically addresses software performance bottlenecks through a feedback-driven approach, iteratively refining code to improve execution speed and resource efficiency while maintaining functional correctness. Our evaluation of SpeedGen on the BigCodeBench dataset demonstrated its ability to consistently improve code performance across a wide range of programming tasks. The results show that SpeedGen not only outperforms traditional LLM-based approaches in terms of average speedup factor, but also achieves a higher success rate in generating correct and optimized code.

SpeedGen's structured pipeline, which includes preprocessing, performance evaluation, code generation, decoding and integrity verification, ensures a robust and reliable optimization process. By incorporating automated profiling and a feedback loop, SpeedGen provides a scalable solution that minimizes manual intervention, thereby streamlining the software development lifecycle. The integration of techniques such as deterministic profiling, cache flushing and CPU core pinning enhances the reliability and accuracy of performance measurements, setting a new standard for LLM-driven code optimization.

Our results suggest that SpeedGen significantly enhances the potential of LLMs in software engineering, paving the way for more efficient and automated performance tuning in diverse programming domains. The adaptability and robustness of the framework highlight its potential as a valuable tool for developers seeking to automate and optimize code performance with minimal manual effort. As LLMs continue to evolve, SpeedGen's approach to leveraging these models for targeted performance improvements represents a promising direction for the future of automated software optimization.

VII. Future Work

While SpeedGen demonstrates significant potential in automating code performance optimization through the use of large language models, some areas remain open for further scientific exploration and enhancements.

Currently, SpeedGen primarily supports Python code optimization. Future work could involve extending the pipeline to support additional programming languages, including languages known for their high performance like C, C++ and Rust [63]. This would widen the applicability of SpeedGen, making it a tool for developers working in different programming ecosystems, especially those programming languages known for being used in performance critical environments.

As new LLMs are developed, integrating more advanced models with enhanced understanding and optimization capabilities could further improve SpeedGen's performance. Also, experimenting with multi-model strategies, where different models contribute to various phases of the optimization process, could lead to even better results [64].

Making SpeedGen more accessible through a user-friendly interface or as a plugin for popular development environments such as VSCode could enhance its usability [65]. This would allow developers to leverage SpeedGen's capabilities directly within their workflows, providing real-time performance suggestions and automated optimizations.

References

[1] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*, 2023.

[2] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents, 2024.

[3] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2024.

Task ID	Baseline	SpeedGen			Llama 3.1		
	Performance [ms]	Performance [ms]	Result	Speedup Factor [%]	Performance [ms]	Result	Speedup Factor [%]
0	40886	14519	Faster	181.60	883	Faster	4530.35
11	14	-	Incorrect	-	13	Faster	7.7
12	23	-	Incorrect	-	-	Incorrect	-
15	18	21	Slower	-14.29	28	Slower	-35.71
20	11087	12425	Slower	-12,07	Incorrect	-	
24	203	201	Faster	0.995	-	Incorrect	-
...	...	...	...	...	...	...	...
650	271	146	Faster	85.62	180	Faster	50.56
...	...	...	...	...	...	...	...
1044	12027	2023	Faster	494.51	12020	Faster	0.058
1046	270	143	Faster	88.81	180	Faster	50
1062	31	-	Incorrect	-	-	Incorrect	-
1074	14	8	Faster	75	12	Faster	16.67
Analysis	-	-	67 Faster 32 Slower 41 Incorrect	Average: **285.19**	-	53 Faster 13 Slower 75 Incorrect	Average: **204.63**

TABLE IV: Evaluation of SpeedGen in terms of execution result, speedup factor and runtime compared to the raw usage of LLama 3.1 without feedback loops.

[4] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[5] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, and Ilge Akkaya et al. Gpt-4 technical report, 2024.

[6] Tung Phung, Victor-Alexandru Pădurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generative ai for programming education: Benchmarking chatgpt, gpt-4, and human tutors, 2023.

[7] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do llms really do their job?, 2024.

[8] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE '07)*, pages 171–187, 2007.

[9] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A survey of performance optimization for mobile applications. *IEEE Transactions on Software Engineering*, 48(8):2879–2904, 2021.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[11] Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. Automatic code generation using pre-trained language models, 2021.

[12] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023.

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[15] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[16] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[17] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174, 2023.

[18] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[19] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers, 2020.

[20] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[21] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[22] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Junnan Li, and Steven Hoi. Codet5mix: A pretrained mixture of encoder-decoder transformers for code understanding and generation, 2023.

[23] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *IEEE Access*, 10:82434–82455, 2022.

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[25] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23, 07 2006.

[26] Delia Velasco-Montero, Jorge Fernández-Berni, and Angel Rodríguez-Vázquez. *Relevant Hardware Metrics for Performance Evaluation*, pages 61–88. Springer International Publishing, Cham, 2022.

[27] Jan Treibig, Georg Hager, and Gerhard Wellein. Performance patterns and hardware metrics on modern multicore processors: Best practices for

performance engineering. In Ioannis Caragiannis, Michael Alexander, Rosa Maria Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, pages 451–460, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[28] Gabriel Campeanu and Mehrdad Saadatmand. Run-time component allocation in cpu-gpu embedded systems. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1259–1265, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Madhura Purnaprajna, Marek Reformat, and Witold Pedrycz. Genetic algorithms for hardware–software partitioning and optimal resource allocation. *Journal of Systems Architecture*, 53(7):339–354, 2007.

[30] Shinobu Fujita, H. Noguchi, K. Nomura, K. Abe, E. Kitagawa, N. Shimomura, and J. Ito. Novel nonvolatile l1/l2/l3 cache memory hierarchy using nonvolatile-sram with voltage-induced magnetization switching and ultra low-write-energy mtj. *IEEE Transactions on Magnetics*, 49(7):4456–4459, 2013.

[31] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on Information Theory*, 60(5):2856–2867, 2014.

[32] Peter Brass. *Advanced data structures*, volume 193. Cambridge university press Cambridge, 2008.

[33] Donghe Kang, Oliver Rübel, Suren Byna, and Spyros Blanas. Predicting and comparing the performance of array management libraries. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 906–915, 2020.

[34] Fernando López de la Mora and Sarah Nadi. Which library should i use?: a metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE '18. ACM, May 2018.

[35] Alexander Chatzigeorgiou and George Stephanides. *Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors*, page 65–75. Springer Berlin Heidelberg, 2002.

[36] Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c#. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14. ACM, May 2014.

[37] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[38] Vasilios Kelefouras and Karim Djemame. A methodology for efficient code optimizations and memory management. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, page 105–112, New York, NY, USA, 2018. Association for Computing Machinery.

[39] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, September 2009.

[40] Sanket Tavarageri, Gagandeep Goyal, Sasikanth Avancha, Bharat Kaul, and Ramakrishna Upadrasta. Ai powered compiler techniques for dl code optimization, 2021.

[41] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: Robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.

[42] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[43] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. 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*, pages 56–63, 2020.

[44] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019.

[45] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. Deepperf: A deep learning-based approach for improving software performance, 2022.

[46] Junjie Ye, Xuanting Chen, Nuo Xu, Can Zu, Zekai Shao, Shichun Liu, Yuhan Cui, Zeyang Zhou, Chao Gong, Yang Shen, et al. A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *arXiv preprint arXiv:2303.10420*, 2023.

[47] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits, 2024.

[48] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation, 2023.

[49] Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models, 2024.

[50] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, Xing Xie, et al. Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization. *arXiv preprint arXiv:2306.05087*, 2023.

[51] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-free llm benchmark, 2024.

[52] Cheryl Lee, Chunqiu Steven Xia, Jen tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R. Lyu. A unified debugging approach via llm-based multi-agent synergy, 2024.

[53] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models, 2024.

[54] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future, 2024.

[55] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Assured llm-based software engineering, 2024.

[56] Python Software Foundation. profile and cprofile module reference. https://docs.python.org/3/library/profile.html#module-cProfile, 2024.

[57] John Gruber. Daring fireball: Markdown. https://daringfireball.net/projects/markdown/, 2004.

[58] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

[59] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Comput. Surv.*, 52(1), feb 2019.

[60] Abhimanyu Dubey et al. The llama 3 herd of models, 2024.

[61] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[62] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Prentice Hall, 2006.

[63] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[64] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024.

[65] Microsoft Corporation. Visual studio code. https://code.visualstudio.com/, 2015.