Compiler-Runtime Co-operative Chain of Verification for LLM-Based Code Optimization

Hyunho Kwon
Yonsei University
Seoul, Republic of Korea
hyunho@yonsei.ac.kr

Sanggyu Shin
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
sg7307.shin@samsung.com

Ju Min Lee
Yonsei University
Seoul, Republic of Korea
jumin@yonsei.ac.kr

Hoyun Youm
Yonsei University
Seoul, Republic of Korea
hoyun@yonsei.ac.kr

Seungbin Song
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
sb15.song@samsung.com

Seongho Kim
Yonsei University
Seoul, Republic of Korea
seongho-kim@yonsei.ac.kr

Hanwoong Jung
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
hw7884.jung@samsung.com

Seungwon Lee
Samsung Advanced Institute of Technology
Suwon, Republic of Korea
seungw.lee@samsung.com

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3315-9288-2/26/$31.00 ©2026 IEEE | DOI: 10.1109/CGO68049.2026.11395240

***Abstract*—Large Language Models (LLMs) have recently shown promise in compiler optimizations such as loop vectorization and memory access restructuring. However, due to their generative nature, LLM-optimized code may contain syntax errors or semantic inconsistencies. While state-of-the-art compilers using LLMs employ symbolic verification to ensure correctness, they fail to fully utilize LLM-based optimizations due to the limited and unreliable verification coverage. This work introduces CoV, a compiler-runtime co-operative Chain of Verification framework that safely integrates LLM-based code transformations into modern compilation workflows. CoV employs a multi-stage verification pipeline that begins with lightweight static checks such as syntax validation and profiling-based checksum filtering, and then applies symbolic equivalence verification using tools like Alive2. For code fragments that cannot be statically verified, CoV inserts runtime verification mechanisms to ensure correctness during execution. These runtime checks are optimized through verification parallelization and batching to minimize overhead. This work implements a prototype CoV framework atop an LLM-based automatic vectorizer within LLVM, and evaluates it using 151 loops in the TSVC benchmark suite and three realistic applications. CoV expands vectorization coverage by 13.9% and 10.6% over LLVM and GCC -O3 vectorization, respectively. In addition, CoV successfully vectorizes loops in three realistic applications that are not handled by the -O3 vectorization.**

***Index Terms*—Large Language Model, Code Optimization, Compiler, Verification**

I. Introduction

Recent advances in Large Language Models (LLMs) [1]–[11] have opened new opportunities in compiler optimization, including loop vectorization, memory access restructuring, and data layout transformation. By learning optimization patterns from large-scale datasets of source code and their corresponding optimized representations, recent LLM-based compilers [12]–[16] support sophisticated code transformations beyond the capabilities of conservative rule-based optimizations or guide optimization heuristics. However, the generative nature of LLMs introduces significant correctness concerns. Unlike conventional compiler optimizations that preserve program semantics, LLMs may generate syntactically valid but semantically incorrect code, potentially violating the original program behavior. Thus, robust correctness checking mechanisms are essential for LLM-based compiler optimization.

To ensure correctness, existing LLM-based compilers [12], [13] often apply static symbolic verification to validate the semantic equivalence between the original and LLM-optimized code. Tools such as Alive2 [17] perform equivalence checking at the intermediate representation (IR) level using SMT solvers. While effective for a subset of well-structured transformations, symbolic verification often fails in practice due to several limitations: lack of support for complex control-flow restructuring, imprecise modeling of pointer aliasing, incomplete coverage of IR constructs, and frequent solver timeouts. Consequently, a large fraction of LLM-based transformations remains unverifiable, leaving compilers unable to safely apply these optimizations despite their potential benefits.

To complement symbolic verification, some LLM-based compilers [12], [13] additionally employ profiling-based validation, which compares the outputs of the original and transformed code on sampled inputs. Since mismatched outputs directly indicate semantic violations, this approach is effective in identifying incorrect transformations. However, profiling-based validation may yield false positives by accepting transformations that happen to produce correct outputs only on the sampled inputs. As a result, while profiling provides an efficient and practical means of filtering out incorrect code, it cannot guarantee the correctness of LLM-based transformations.

To overcome the limited coverage of symbolic and profiling-based verification, this work proposes runtime verification for LLM-optimized code verification. The runtime speculatively executes the LLM-optimized code that cannot be statically verified while concurrently executing the original code in a separate process. After both executions complete, the runtime compares their outputs to ensure consistency between the two codes. If the results match, the runtime continues execution along the LLM-optimized path, thus realizing performance gains. If mismatched, the system adopts the original path, discarding the speculative result on the LLM-optimized path.

By incorporating the newly proposed runtime verification with existing static methods, this work presents CoV, a compiler-runtime co-operative Chain of Verification framework that enhances correctness through a staged verification pipeline. CoV first applies lightweight static verification such as syntax checks and profiling-based checksum filtering to quickly discard invalid transformations. It then applies symbolic equivalence checking using Alive2 for transformations that can be formally verified. For the remaining unverifiable code fragments, CoV inserts runtime verification logic that speculatively executes the optimized code and dynamically compares its behavior against the original version during execution.

To demonstrate the applicability of CoV, this work implements a prototype compiler targeting LLM-based loop vectorization atop the LLVM infrastructure [18]. The compiler identifies vectorization candidates via source-level annotations and first attempts transformation using LLVM's vectorizer. For loops not statically vectorized, CoV invokes an external LLM-based vectorizer to generate LLM-optimized code. The generated code then passes through the CoV verification pipeline. Transformations that pass static verification are compiled directly, while unverifiable ones are instrumented with runtime verification logic. The CoV runtime speculatively executes the LLM-optimized loop in parallel with the original version, comparing memory writes to ensure semantic equivalence.

This work evaluates CoV prototype using the Test Suite for Vectorizing Compilers (TSVC) benchmark suite [19] and three realistic applications, employing DeepSeek-R1 671B model [1] as the underlying LLM-based vectorizer. Out of 151 loop kernels in the TSVC benchmark, LLVM and GCC -O3 optimization pipeline statically vectorizes 67 and 81 loops. For the remaining loops that -O3 fails to vectorize, CoV applies LLM-based vectorization and statically verifies 8 and 6 additional loops, while further verifying 13 and 10 statically unverifiable loops using runtime verification. CoV achieves a 1.18× geomean speedup over the state-of-the-art static-verification-only configuration [12], [13], which corresponds to 1.67× and 1.48× speedups over LLVM and GCC -O3 vectorization, respectively. To demonstrate the applicability of CoV, this work additionally applies CoV to realistic applications such as BlackScholes [20], Kmeans [21], and Lattice Boltzmann Method (LBM) [22]. In these workloads, CoV successfully enables LLM-based vectorization of loops that are not optimized by LLVM and GCC -O3 optimization pipeline, with correctness ensured through runtime verification.

TABLE I: Comparison of LLM-based Compilers

LLM Compilers	Optimization	Verification
LLM Compiler [14]	Code size	No verification required
LLM-Vectorizer [12]	Vectorization	Symbolic only
VecTrans [13]	Vectorization	Symbolic only
This work	Vectorization	Symbolic and Runtime

The key contributions of this work are as follows:

- A compiler-runtime co-operative Chain of Verification framework, CoV, that combines static verification such as syntax checking, profiling-based validation, and symbolic verification with runtime verification, enabling trustworthy deployment of LLM-based optimizations.
- A runtime verification mechanism that expands the verification coverage of LLM-based optimizations beyond the limits of static verification.
- A prototype implementation of CoV for loop vectorization, built on LLVM and integrated with a 671B-parameter LLM-based vectorizer, which shows vectorization coverage and performance improvement on the TSVC benchmarks suite and three realistic applications.

II. Motivation

A. Large Language Models for Compiler Optimization

Machine learning [23]–[30], especially Large Language Models (LLMs) [12], [13], have recently emerged as promising tools for overcoming limitations of conservative compiler optimization. Conventional compiler optimizations are rule-based, ensuring correctness by avoiding transformations in the presence of potential ambiguity like uncertain memory aliasing. While this cautious approach guarantees semantic preservation, it prevents the compiler from leveraging numerous optimization opportunities, especially in complex or irregular loop structures. On the other hand, LLMs can learn optimization strategies from large-scale datasets of code and their corresponding transformations, and propose aggressive optimizations that often bypass the constraints of traditional rule-based compilers. This flexibility allows LLMs to expose optimization opportunities that static analyses often miss.

Several recent studies have explored this potential in compiler contexts as described in Table I. LLM-Vectorizer [12] and VecTrans [13] employ LLMs to rewrite scalar loops into vectorizable forms by injecting SIMD intrinsics or restructuring source-level loop patterns. These systems extend vectorization opportunities beyond the reach of standard compiler backends, without requiring backend modifications. LLM Compiler [14] leverages LLMs to recommend compiler flags that reduce binary size and explores compiler emulation via prompt-based code completion. These systems illustrate how LLMs can contribute to both the transformation phase and the tuning decision stage of the compilation process.

Despite their promise, LLM-based compiler optimizations face a fundamental challenge: the lack of correctness guarantees. Since LLMs operate without formal analysis or transformation

```
1 for (int i = 0; i < LEN_2D; i++){
2   for (int j = 0; j < LEN_2D; j++)
3       aa[j][i] = aa[j][i] + bb[j][i] * cc[j][i];
4   a[i] = b[i] + c[i] * d[i];
5 }
```

Fig. 1: TSVC benchmark example successfully vectorized by CoV, but not by the standard LLVM -O3 optimization pass.

rules, their outputs are inherently unreliable. LLM-optimized code may be syntactically valid, but may fail to preserve the semantics of the original code, introducing correctness violations. This lack of correctness assurance poses a major barrier to adopting LLM-based optimization.

B. Static Verification for Program Semantics

To overcome the lack of formal correctness guarantees of LLM-based optimization, validating that transformed code preserves the semantics of the original program is essential. Existing LLM-based compilers [12], [13] adopt static symbolic verification, using tools like Alive2 [17] to check semantic equivalence at the LLVM IR level. Alive2 translates both the original and optimized code into logical formulas, and formally proves their equivalence using an SMT solver. Thanks to the formal proof, the compilers can ensure that the LLM-optimized code is identical to the original code.

While powerful in principle, Alive2 suffers from several critical limitations in practice. First, certain language features such as architecture-specific intrinsics are either unsupported or only partially modeled. Second, as the complexity of the IR increases, SMT solving becomes computationally expensive, frequently causing timeouts or memory exhaustion errors. Finally, Alive2 is not robust to small changes, as even semantically benign optimizations may lead to verification failures, despite the original version having previously passed validation. These issues make Alive2 unsuitable for verifying many LLM-based transformations, particularly those involving noncanonical IR patterns or aggressive restructuring.

As an alternative to Alive2, KLEE [31], a symbolic execution tool, can be used for LLM-optimized code verification. KLEE detects behavioral differences by automatically generating test inputs that explore execution paths. However, KLEE also suffers from scalability limitations. As the number of branches grows, the number of possible paths increases exponentially, making path coverage infeasible in many cases.

These limitations highlight the need for complementary runtime verification techniques that can guarantee identical execution of LLM-optimized code when static analysis proves insufficient or inapplicable.

C. Application of LLM-based Optimization: Vectorization

Vectorization is a widely used optimization that improves performance by exploiting data-level parallelism in loops. However, compilers often miss vectorization opportunities due to conservative analysis rules that aim to preserve program correctness. Our evaluation on 151 loops in the TSVC benchmark suite [19] shows that LLVM -O3 optimization pass statically vectorizes only 44.4% of the loops. This limited coverage stems primarily from the compiler's inability to disambiguate memory dependencies and aliasing relationships. Figure 1 illustrates a loop example that LLVM -O3 optimization pass fails to vectorize. The loop is vectorizable if `aa`, `bb`, and `cc` are allocated in distinct memory regions. However, the compiler fails to prove that these memory regions are independent due to conservative alias analysis. As a result, it skips loop vectorization, even though the transformation would be safe and beneficial for most cases.

The LLM-based optimization successfully vectorizes the loop, and achieves a 3.5× speedup over the -O3 baseline on the same example. This demonstrates that LLMs can unlock optimization opportunities that conventional compilers miss, offering a practical path to improved performance. However, the lack of formal correctness guarantees in LLM-based optimization presents a major obstacle. In the above case, the transformed loop passed profiling-based checksum validation, yet Alive2 failed to prove semantic equivalence, returning an inconclusive result due to SMT solver limitations.

To address this, LLM-Vectorizer [12] applies various code transforms such as loop unrolling, memory alias annotations, and manual alignment enforcement to make LLM-transformed code verifiable. Despite these efforts, approximately 20% of the evaluated TSVC benchmarks [19] still could not be verified by Alive2. Without reliable and automated validation mechanisms, developers are forced to either manually inspect and reason about the correctness of LLM-optimized code or abandon its use altogether. To safely integrate LLM-based transformations into compiler workflows, a supporting framework is needed that provides correctness guarantees while preserving the performance potential of LLM-optimized code.

III. CoV Design

This paper presents CoV, a compiler-runtime co-operative Chain of Verification framework that enables a broader range of compiler optimization through LLM-based optimizations while ensuring the correctness of the code. CoV consists of two key features, CoV compiler and runtime library, which allow static and runtime verification. The CoV framework provides fully-verified LLM-optimized code for code that is not optimizable in traditional compilers. In this work, we primarily target loop vectorization as our optimization target, a domain where conventional compilers often miss opportunities due to conservative analysis and strict transformation rules.

A. Overview of CoV Framework

Figure 3 illustrates the overview of CoV framework, consisting of two major components, CoV compiler and runtime library. CoV compiler takes a C program as input, in which vectorization target loops are explicitly annotated with pragmas at the source level as exemplified in Figure 2. The compiler frontend parses these pragmas and identifies the corresponding loops as candidate regions for vectorization. CoV first attempts to vectorize each candidate loop using a conventional static

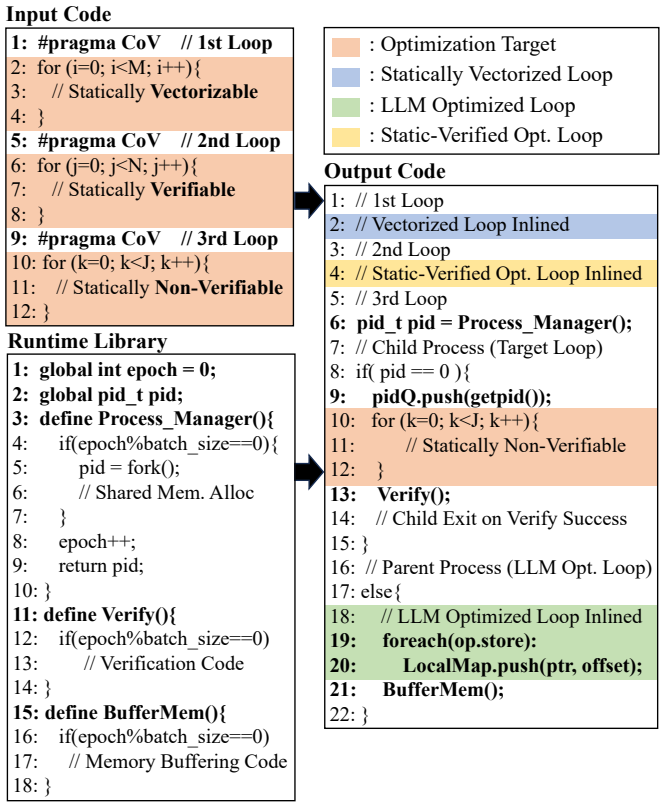


Fig. 2: The input code example shows three types of code that can be optimized by CoV: statically vectorizable code, statically verifiable code, and statically non-verifiable code.

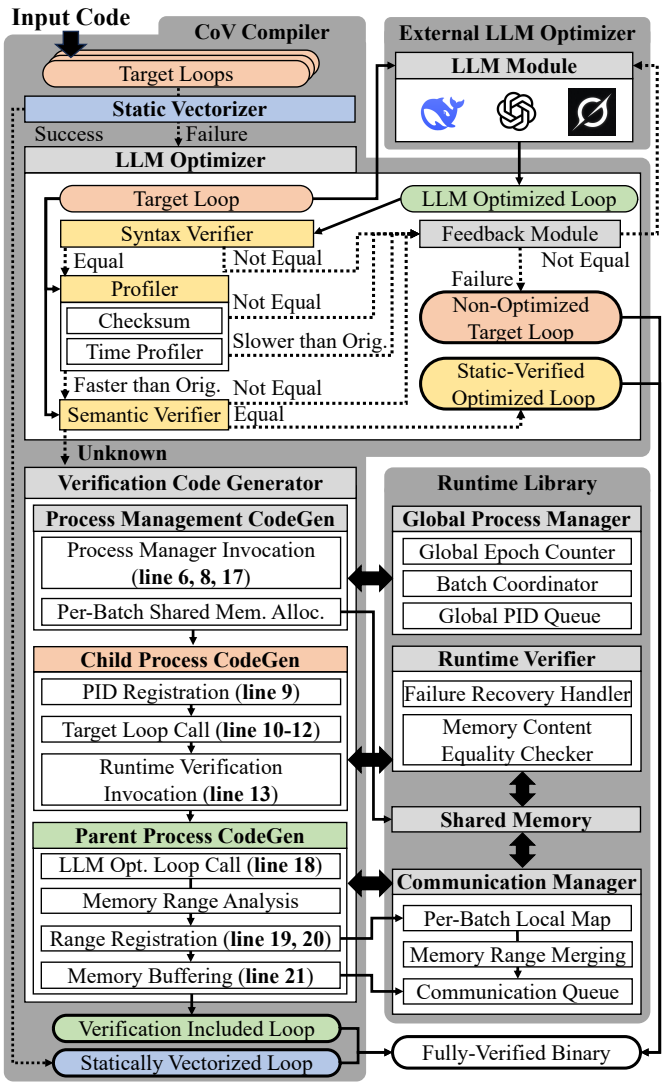


Fig. 3: Overview of CoV Framework, including CoV compiler, LLM optimizer, and runtime library to support both static and runtime verification for LLM-optimized codes. Line denotes a code line from the Output Code in Figure 2.

vectorizer. If the static vectorizer successfully applies compiler-supported vector transformations, the compiler uses this version directly, bypassing the LLM optimization pipeline. This avoids unnecessary compilation and verification overhead for code regions that are already well-optimized under conventional transformation rules.

In contrast, target loops that are not vectorized by the static vectorizer are forwarded to the LLM Optimizer, which generates vectorized LLVM IR based on prompt-driven interaction with LLMs. As the generated IR is not guaranteed to be semantically equivalent to the original, it undergoes a three-stage static verification pipeline, syntax verification, profiling-based output validation, and semantic verification, which will be discussed in detail in Section IV. If verification fails (i.e., a syntax or semantic mismatch), CoV iteratively refines the prompt and re-queries the LLM with additional diagnostic context, repeating this cycle until a verified transformation is obtained or an iteration threshold is reached.

In cases where the static verifier produces an inconclusive result, such as when Alive2 returns inconclusive results due to SMT solver timeouts or unsupported constructs, CoV falls back to runtime. The runtime verification code generator inserts runtime library functions that enable speculative parallel execution of both the LLM-optimized loop and its baseline counterpart. During the execution, memory writes from the LLM-optimized loop are buffered and subsequently compared against those from the baseline loop to detect any behavioral deviation. This runtime comparison, combined with speculative execution, complements the static verification pipeline and enables CoV to ensure correctness even in cases where formal verification tools yield inconclusive results.

B. Runtime Verification Workflow

When a loop requiring runtime verification is encountered during execution, the system triggers a fork through an inserted invocation, as illustrated in Figure 2. This creates a child process, enabling parallel execution of the original and optimized code. While the parent process executes the LLM-optimized version of the loop, the child process executes the original loop, followed by verification for the target loop. During execution, instrumentation code inserted alongside each store instruction records the destination address and size of the memory write. After the optimized loop completes, the parent

process accesses the values stored at the recorded memory addresses and copies them into a shared memory pool. For each entry, metadata, including the original address, the size of the store, and the offset within the shared memory pool, is committed and pushed to a communication queue. This metadata enables the child process to locate and compare the stored values during the subsequent verification phase. After this step, the parent process continues executing speculatively beyond the verification region, without waiting for verification to finish. This design allows runtime verification to proceed in parallel, thereby reducing runtime delay and avoiding execution stalls typically caused by synchronous verification.

In contrast, the child process executes the baseline (unoptimized) version of the same function in parallel. Once the unoptimized execution completes, the child process reads the metadata from the communication queue and performs verification by comparing the contents of its own memory space against the reference data stored in the shared memory pool. For each memory region recorded, the child process checks whether the same value is stored in the same location. If all comparisons succeed, the optimized version is deemed semantically equivalent to the baseline, and the child process exits. However, if any mismatch is detected, indicating a semantic divergence, the child process terminates the parent process to prevent incorrect program behavior and continues execution from the verified baseline state.

IV. CoV Compiler

A. Static Vectorizer

The CoV compiler is implemented on top of the LLVM infrastructure [18]. As a first step, it applies the -O3 optimization level to the input loop and uses LLVM reporting flags to determine whether vectorization has been successfully applied to the annotated target loop. If the static vectorizer succeeds, the compiler bypasses both the LLM-based optimization and runtime verification code generation stages, treating it as a statically vectorized loop. The resulting code is then incorporated into the final binary without further transformation. In contrast, functions that fail to be vectorized at this stage are forwarded to the LLM Optimizer for further processing.

B. LLM Optimizer

LLM Optimizer operates on loops that are not vectorized by the static vectorizer. It submits the target loop to the LLM module with a prompt instructing it to perform vectorization. The LLM-optimized code then passes through a three-stage static verification pipeline. The first stage is a syntax verifier, which uses the LLVM verify option to ensure syntactic correctness. If successful, the code proceeds to the profiler, which performs a coarse-grained functional equivalence check using checksums and evaluates performance via execution time comparison. The time profiler filters out poorly optimized code by rejecting LLM-optimized code that is predicted to be slower than the original, returning to the feedback module. If both profiling checks succeed, the code is passed to the semantic verifier, implemented with Alive2 [17]. Semantic verifier attempts to prove semantic equivalence between the original and transformed LLVM IR by encoding them into SMT formulas [32] and solving for equivalence. If the verification succeeds, the LLM-optimized loop is considered a static-verified optimized loop and compiled into the final binary.

If any stage in the pipeline fails, an error code is sent to the feedback module, which automatically constructs a new prompt and resubmits the transformation request to the LLM. This iterative refinement process continues until either a valid vectorized loop is produced or a user-defined iteration limit is reached. If no valid transformation is found within the allowed iterations, the compiler deems the loop non-vectorizable and defaults to the original unvectorized version. In cases where Alive2 returns an inconclusive result, the loop is marked as unknown, and CoV transitions to the runtime verification stage to validate the transformation during execution.

C. Runtime Verification Code Generator

When static verification fails to conclusively validate a target loop, CoV compiler invokes the runtime verification code generator. This component instruments the loop with calls to runtime library functions that coordinate speculative execution and memory-based equivalence checking. The output code in Figure 2 shows runtime verification code generated by the code generator, with all subsequent line numbers referring to this code. Figure 4 further illustrates the three-step transformation applied during runtime verification code generation.

Process Management: The code generator begins by inserting an invocation to the process manager, which facilitates the concurrent execution of both the target and the LLM-optimized loop in separate processes (Line 6). Specifically, the parent process executes the LLM-optimized loop, while the child runs the baseline version (Lines 10–12, 18–20). To enable memory comparison, each store instruction within the LLM-optimized loop is instrumented to log the destination address and size of the memory write to a local map via `LocalMap.Push` (Line 20). In addition, to ensure full coverage, the CoV compiler also analyzes intrinsics such as `llvm.masked.store`, identifying functions in which store operations occur internally and logging memory addresses passed as arguments when such patterns are detected. Once loop execution completes, the parent process invokes the `BufferMem` function (Line 21), which at runtime: (1) merges overlapping store address ranges collected in the local map, (2) copies the associated memory contents into a shared memory pool, and (3) pushes metadata, consisting of base address, size, and pool offset, into a communication queue. In the child process, once the baseline loop execution finishes, the verifier is called to compare the memory contents written by the child process against those stored by the parent, using the metadata communicated via the queue (Line 13).

CoV compiler also modifies the code to defer output operations (e.g., I/O and system calls) in the speculative execution path until the runtime verification completes. On verification failure, the deferred operations are discarded and never committed, ensuring the observable program behavior

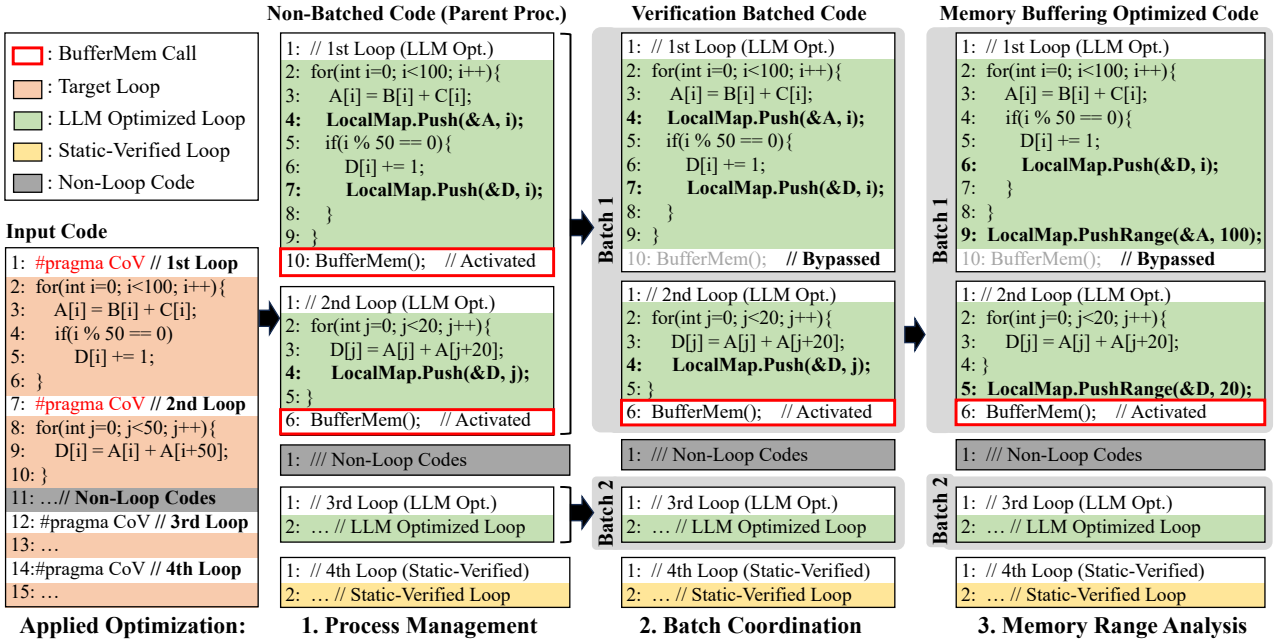


Fig. 4: Demonstration of the three-step runtime verification code generation through process management, batch coordination, and memory range analysis.

remains consistent. Similarly, CoV compiler excludes loops that include input operations from its optimization target selection to preserve the program output.

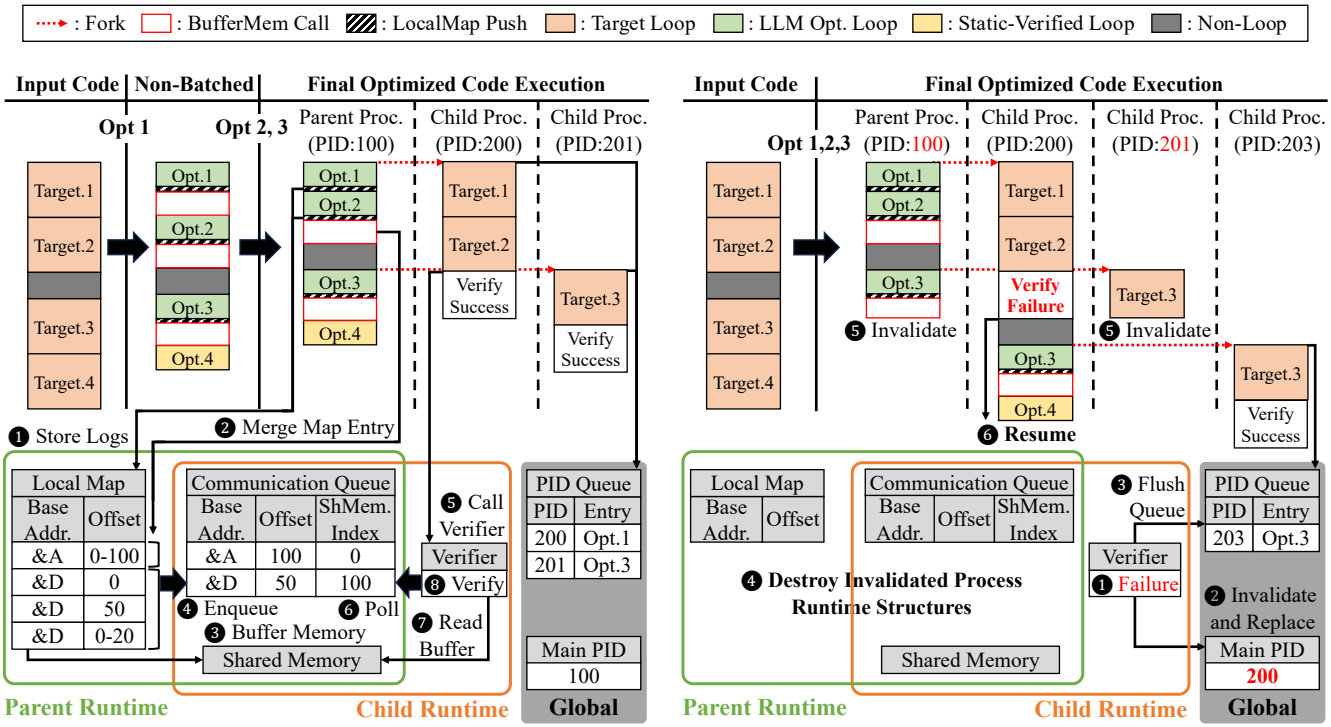
Batch Coordination: When verification targets appear frequently, or target loop invocations occur inside other loops, naively invoking a fork at each verification point can incur substantial overhead. This overhead stems from the cumulative cost of process creation, memory duplication, and page faults due to Copy-on-Write behavior. To reduce this cost, CoV employs an epoch-based batching mechanism, where a batch size determines how many target loops are grouped into a single verification batch. Under this scheme, a single fork is issued per batch, spawning one parent and one child process to execute all loops within the batch. For example, as shown in Figure 4, if the epoch is set to 2, targets 1 and 2 form a batch. The fork occurs before the first target, and the parent and child processes sequentially execute the LLM-optimized and baseline loops, respectively. Runtime verification, including buffering, and comparison, is deferred until the final target in the batch completes. All memory logs are merged and compared in a single verification phase, amortizing the cost of process management and inter-process communication.

Memory Range Analysis: Even with batching, store instructions inside loops may execute on every iteration, resulting in excessive logging overhead if each store is recorded individually. To mitigate this, CoV compiler performs static store range analysis to conservatively determine the memory regions written by each store instruction. It leverages the LLVM Scalar Evolution (SCEV) pass to analyze `getelementptr` (GEP) expressions used in computing store addresses. For each store instruction, the compiler identifies the base address and computes the maximum offset accessed across all loop iterations. When the analysis succeeds, the compiler inserts a single call to `LocalMap.PushRange` at the end of the loop, recording the full store range in the local map. This reduces the number of log entries pushed into the map and simplifies the subsequent merge operation in `BufferMem`.

V. CoV Runtime

A. Runtime Verification

Runtime verification begins when the process manager is invoked. The process manager initially checks the global epoch counter to determine whether the current loop is the start of a new verification batch. In that case, a fork is issued to execute the LLM-optimized and baseline loops concurrently. At the same time, a shared memory region is allocated to enable inter-process communication between the parent and child processes. The sequence of runtime verification is illustrated in Figure 5a with eight steps. In the parent process, runtime library calls inserted by the compiler capture memory store operations at pre-identified logging points. Each store logs the destination address and offset into a local map during loop execution (step ❶). Once all LLM-optimized loops in the batch complete, the parent merges the logged store entries to form a non-overlapping list of memory ranges (step ❷). This is achieved by sorting the entries by base address and coalescing contiguous or overlapping ranges based on offset comparisons. For each merged store range, the parent buffers the corresponding memory contents into a shared memory pool (step ❸). As each range is buffered, the associated metadata, including base address, offset, and index within the shared memory pool, is pushed into a communication queue to notify the verifier (step ❹). After all ranges are processed, a finish signal is enqueued into the queue, and the parent resumes normal execution beyond the verification region.



(a) Runtime verification sequence with verification success.

(b) Failure handling procedure where verification of batch 1 fails.

Fig. 5: Runtime example of CoV framework with optimized code from Figure 4. Runtime structures for the first batch are illustrated at the bottom to describe interactive runtime library operations.

Meanwhile, the child process registers its PID into a PID queue at initialization and begins executing the baseline loop. Upon batch completion, the verifier is invoked (step ❺) and polls the communication queue (step ❻) to retrieve range metadata and perform comparisons. It then reads the corresponding buffers from the shared memory pool (step ❼), and compares them against its own memory space (step ❽). This process iteratively continues until the finish signal is received from the parent process. If all comparisons succeed, the verifier checks whether the current process PID matches the head of the global PID queue. If so, the process dequeues itself and terminates. Otherwise, it continues polling until it becomes the head, ensuring correct synchronization across batched verifications. This coordination ensures that verification for earlier batches completes before later processes exit, preserving the integrity of speculative execution and verification.

B. Verification Failure Handling Procedure

Verification failure handling procedure begins only after confirming that the current process is at the head of the global PID queue, ensuring that the previous batch verification procedure has completed. This synchronization prevents premature failure handling from interfering with earlier verification phases. Figure 5b illustrates the failure handling procedure invoked by a verification failure in the first batch (step ❶). The procedure begins by retrieving the parent process PID from the global main PID and issuing a kill signal to invalidate it. The PID of the process that detects the failure then replaces the global main PID, taking over as the new parent (step ❷). Since the LLM-optimized code has been deemed incorrect, any remaining processes in the global PID queue spawned for future batch verifications are no longer meaningful. Thus, the procedure selects every process in the global PID queue as an invalidation target to prevent further execution of invalid code paths, flushing the queue for invalidation (step ❸). For each invalidation target, associated runtime structures are destroyed, and the shared memory region is unmapped (step ❹), safely invalidating the process (step ❺). Finally, the process registered as the global main PID resumes execution and continues runtime verification (step ❻). This approach ensures clean recovery from semantic mismatches while preserving correctness and enabling continued speculative execution across future batches.

VI. Evaluation

All experiments are conducted on a machine equipped with an Intel Xeon Gold 6326 CPU (32 cores, 2.9GHz) and 512GB DDR4 memory, running Ubuntu 22.04. The CoV compiler is implemented on top of LLVM Clang 20.0.1 version [18]. For LLM-based optimization, we use DeepSeek-R1, a 671B-parameter reasoning-centric model, served on a dedicated server with 16 NVIDIA H100 GPUs. No fine-tuning or model modification is performed. Comparing with the LLVM and GCC 14.1 [33] -O3 vectorization, we evaluate our framework on 151 loop kernels from the TSVC benchmark suite [19], as well as three realistic applications, including BlackScholes [20], Kmeans [21], and the Lattice Boltzmann Method (LBM) [22].

TABLE II: TSVC Classified by Vectorization and Verification

Loop Category	LLVM		GCC	
	Count	Cum. (%)	Count	Cum. (%)
Statically-Vectorized	67	44.4	81	53.6
Static-Verified Optimized	8	49.7	6	57.6
Runtime-Verified Optimized	13	58.3	10	64.2
Unvectorized	63	–	54	–

We use the reference datasets provided by each benchmark for profiling at compile time, and use randomly generated synthetic datasets for the evaluation to test generalizability across varying data distributions.

For fair comparison, the evaluation includes all the TSVC loops used in the prior work such as LLM-Vectorizer [12], which are inherently amenable to vectorization, but are often left unoptimized by conventional compilers due to conservative analysis. We acknowledge that evaluating only such workloads is a limitation, as less vectorizable loops may result in higher runtime verification failure rates and more limited speedups. We leave the more aggressive vectorization of a broader range of loops to future work involving context-sensitive optimizations with more advanced LLMs.

A. TSVC Benchmark

Table II presents the evaluation results of the CoV framework on the TSVC benchmark suite [19]. We classify the loops into four categories:

- Statically-Vectorized Loop: Loops successfully vectorized by the -O3 pipeline and verified via static analysis.
- Static-Verified Optimized Loop: Loops that are not vectorized by the -O3 pipeline, but for which the LLM-based vectorized version passes static verification.
- Runtime-Verified Optimized Loop: Loops that fail static verification but successfully pass runtime verification.
- Unvectorized Loop: Loops that fail to be vectorized either by static or LLM-based approach, including cases where the LLM exceeds the refinement iteration limit.

CoV vectorizes 21 and 16 more loops than LLVM -O3 and GCC -O3, respectively, on the TSVC benchmark suite, demonstrating its ability to uncover optimization opportunities missed by conventional compilers. Compared to LLVM, the proposed LLM-based pipeline statically verifies and optimizes 8 additional loops (5.3%), and the runtime verification extends coverage by another 13 loops (8.6%). Against GCC, CoV achieves 6 additional loops (4.0%) via the static verification and 10 loops (6.6%) through the runtime verification. Among the 21 loops vectorized by CoV over LLVM, five (s1161, s272, s176, s235, and s314) are statically vectorized by GCC. Nevertheless, CoV enables correct vectorization beyond what static analysis alone can achieve in both compiler baselines.

Analysis of CoV-Verified Optimized Loops: This expanded coverage directly translates into meaningful performance gains. As shown in Figure 6, CoV achieves geometric mean speedups of 1.67×, 1.48× and 1.18× over LLVM -O3, GCC -O3 and the state-of-the-art static-verification-only configuration [12], [13]. By integrating LLM-based optimization with principled correctness guarantees, CoV unlocks previously unreachable optimization opportunities without compromising semantics.

For s1161 and s272, CoV shows lower performance improvement than GCC due to the runtime verification overhead. For the others, CoV outperforms GCC because LLM-based optimization exploits more aggressive transformations such as loop distribution, and LLVM intrinsics such as `fmuladd`. GCC suffers from degraded or limited speedup for s176, s235, and s314, while CoV achieves meaningful performance improvements, showing that conventional vectorization leaves residual optimization opportunities that CoV is able to capture.

In benchmarks like s231, s2275, and s235, traditional compilers miss vectorization opportunities due to loop-carried dependencies and non-perfect nested loops. The LLM addresses these issues by applying transformations such as loop interchange or distribution, reordering computations to isolate independent components, as shown in Figure 9. These transformations enable vectorization by removing or bypassing dependency cycles that static analysis cannot safely disprove. In s257 and s322, where apparent data dependencies block vectorization, the LLM speculatively rewrites expressions to pre-compute or isolate reused values. By lifting invariant computations and restructuring memory accesses, the transformation reduces dependency visibility. Similarly, in s212, statement reordering and the introduction of temporaries eliminate subtle cross-iteration dependencies that defeat static heuristics.

Control flow within loops often prevents vectorization due to the need for uniform execution paths. In s272 and s482, the LLM replaces conditional branches with predicated operations using vector masks. This allows the compiler to emit SIMD instructions even when the original loop includes data-dependent conditionals. Certain benchmarks require a structural rethinking of the loop index space. For example, s281 involves reverse-order memory accesses that inhibit forward vectorization. The LLM overcomes this by using vector permutation operations such as shufflevector, enabling effective SIMD utilization even with non-linear access patterns. In s1113, the loop includes invariant scalar operands accessed across all iterations. The LLM exploits this by broadcasting the invariant value across the vector lanes, replacing what would otherwise be sequential scalar loads.

In contrast, there are cases such as the s1161, s1113, and s254 loops where performance degrades due to the overhead introduced by runtime verification. In the case of s1161 and s1113, the LLM-optimized loop does not offer significant performance gains over LLVM or GCC -O3, making the verification overhead relatively more impactful and ultimately resulting in slower execution. For s254, performance degradation occurs due to the presence of a store instruction that could not be analyzed by the memory range analysis, leading to increased logging cost. Nevertheless, across the TSVC benchmark, the number of loops where runtime verification enables both correctness and performance improvement outweighs the cases with overhead-induced slowdowns.

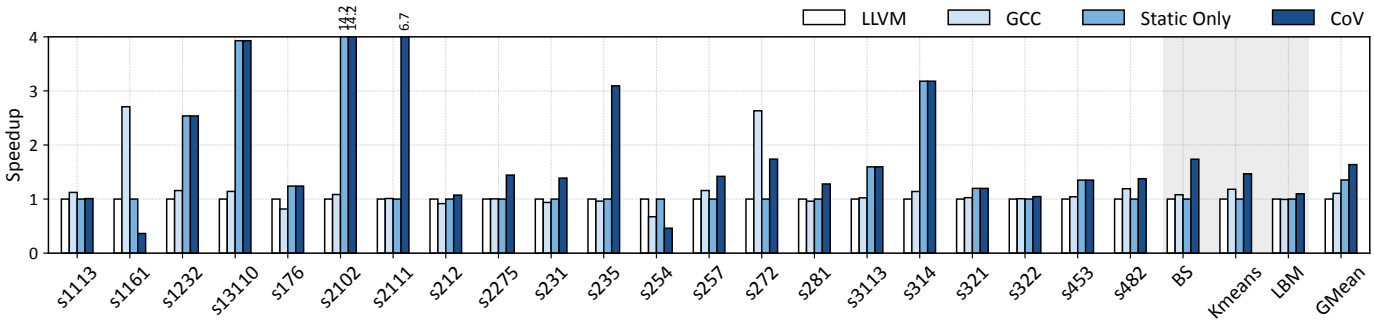


Fig. 6: Speedup achieved by the CoV on TSVC and realistic applications. Realistic applications are highlighted with a shaded gray region. BS denotes BlackScholes. Bars labeled Static Only represent loops that were successfully verified by static analysis.

B. Realistic Applications

We evaluate the practicality of CoV on three realistic applications: BlackScholes, Kmeans, and LBM. CoV vectorizes loops that LLVM and GCC -O3 cannot vectorize, and static verification fails to verify under LLM-based vectorizations. As shown in the shaded region of Figure 6, CoV achieves speedups of 1.74×, 1.47×, and 1.10× for BlackScholes, Kmeans, and LBM, respectively, compared to LLVM -O3, while ensuring correctness via runtime verification. Relative to GCC -O3, CoV attains speedups of 1.61×, 1.24×, and 1.11×. LBM yields a relatively smaller speedup, as its memory-bound kernels spend most cycles on loads and stores rather than computation, limiting the benefits of vectorization.

In BlackScholes, most execution time is spent in the `BlkSchlsEqEuroNoDiv` function, which is repeatedly invoked within inner loops. To enable vectorization, CoV inlines this function and systematically replaces math library calls (e.g., `log`, `sqrt`, and `exp`) with LLVM intrinsics. For Kmeans, CoV targets the `find_nearest_point` kernel, a reduction case where vectorization is typically inhibited by aliasing and by the non-associativity of floating-point arithmetic, since reordering operations can lead to different numerical results. Despite these challenges, CoV applies vectorization beyond conventional compiler heuristics and employs runtime verification to ensure semantic equivalence. In LBM, the `LBM_performStreamCollideTRT` function contains complex control-flow branches and irregular memory accesses due to its array-of-structures layout, which typically blocks vectorization. CoV restructures control flow into predicated select operations and enables vectorization even in the presence of irregular memory patterns by relying on runtime verification to ensure correctness.

Although this work applies CoV to other programs in the PARSEC [34], Rodinia [35], and SPEC CPU 2017 [36] benchmark suites, the current LLM can successfully analyze and vectorize only BlackScholes, Kmeans, and LBM due to context-length limitations. However, this is a limitation of the current LLMs, not of the proposed verification framework. We expect that future models with longer context windows will further extend CoV's applicability.

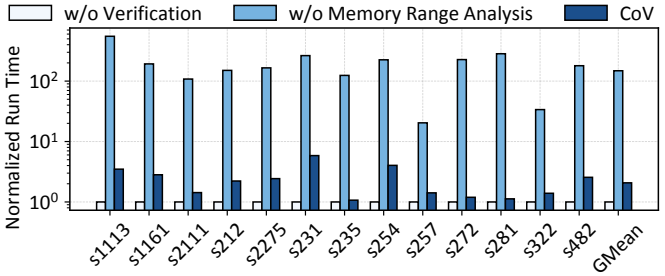


Fig. 7: Effectiveness of memory range analysis, with run time normalized to execution without runtime verification.

TABLE III: Runtime verification overhead breakdown (ms). Ref. time, Alloc, Buf.Mem, LM.Push denotes the execution time without runtime verification overhead, shared memory allocation, BufferMem, and LocalMap.Push.

App.		Ref. time	Runtime Verification Overhead				Total time
			Alloc	Fork	Buf.Mem	LM.Push	
BS	Naïve	9515	20	338	11268	3233	24374
	CoV		3	38	56	2	9614
Kmeans	Naïve	3371	299	10940	11310	7365	33285
	CoV		19	1934	16	22	5362
LBM	Naïve	219042	171	23439	3078905	1657543	4979100
	CoV		1	162	2485	202	221892

C. Reducing Runtime Verification Overhead

Although the CoV framework achieves notable performance gains through LLM-based vectorization, runtime verification inevitably introduces overhead due to shared memory allocation, process forks, and memory instrumentation. To assess the cost and mitigation of this overhead, Figure 7 presents the normalized execution time across three configurations: (1) w/o verification, where optimized loops are executed without any runtime checking; (2) w/o memory range analysis, where verification is enabled but logs all store operations; and (3) CoV, which performs optimized, batched logging based on static memory range analysis. Overall, memory range analysis significantly reduces runtime overhead across TSVC benchmarks. A similar trend is observed in realistic applications, as shown in Table III, where the memory range analysis and batching significantly reduce the dominant costs of runtime verification. The memory range analysis logs memory write addresses once per loop rather than per write, reducing the overhead

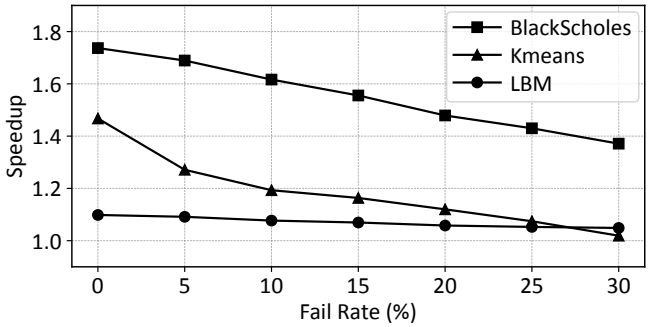


Fig. 8: Speedup drop from runtime verification failures.

```
1  // Original C Code
2  for (int i = 0; i < LEN_2D; i++){
3    a[i] += b[i] * c[i];
4    for(int j = 1; j < LEN_2D; j++)
5      aa[j][i] = aa[j-1][i] + bb[j][i] * a[i];
6  }
7  // LLM-optimzed C Code
8  for (int i = 0; i < LEN_2D; i++)
9    a[i] += b[i] * c[i];
10 for (int j = 1; j < LEN_2D; j++)
11   for (int i = 0; i < LEN_2D; i++)
12     aa[j][i] = aa[j-1][i] + bb[j][i] * a[i];
```

Fig. 9: TSVC example of an original loop and its LLM-optimized version.

of `BufferMem` and `LocalMap.Push`. The batching verifies multiple loops within the same process, avoiding per-loop process forking and reducing initialization costs such as shared memory allocation and fork overhead. CoV reduces average execution time by over an order of magnitude compared to the naïve configuration, while preserving correctness. These results demonstrate that such memory range analysis and batching are crucial for the practical deployment of runtime verification in compiler-integrated optimization pipelines.

D. Runtime Verification Fail Overhead

Figure 8 illustrates how runtime verification failures lead to a drop in speedup across realistic applications. Since profiling-based filters such as checksum comparison effectively exclude potentially incorrect transformations before runtime, no runtime verification failures occurred in our evaluation. To analyze the potential overhead of verification failure and handling procedure, we artificially inject errors into realistic applications at varying rates and measure a drop in speedup, as shown in Figure 8. The runtime verification failure handling procedure itself incurs only a negligible overhead on the order of milliseconds. The larger slowdown observed under enforced failures simply reflects the memory comparison costs, which are normally hidden when verification succeeds. Meanwhile, Kmeans exhibits a relatively large slowdown due to the high overhead of frequent verification on a short-loop target.

E. Compile-Time Overhead

The compile-time of CoV is dominated by (1) the inference time of LLMs, which generates candidate optimizations, and (2) the static verification time of the semantic verifier. The LLM inference time increases when the LLM fails to produce syntactically and semantically correct candidates in early attempts, as this leads to additional refinement iterations that are more likely to occur with complex loops. We use Alive2 for static verification with a one-minute timeout, following prior work [17] showing that longer time limits yield marginal coverage gains at disproportionate compile-time cost. On the TSVC benchmark suite, the average compile time per loop is about 14 minutes. The refinement process requires 1.76 iterations on average, completing within three iterations. For the realistic applications, the average compile-time increases to 26 minutes with 2.78 refinement iterations on average. This increase reflects the greater structural complexity of the loops in realistic applications compared to microbenchmarks of TSVC, where intricate transformations prolong LLM inference and demand additional refinement iterations. Although CoV effectively manages the loops in three realistic applications, extending the same workflow to substantially larger kernels reveals an additional practical difficulty. In such cases, additional refinement iterations accumulate a growing amount of contextual information in the prompt. As the prompt expands, it approaches the model's effective context capacity, eventually resulting in inference failures once the token limit is exceeded.

VII. Discussion

While this work applies CoV to LLM-based loop vectorization, the applicability of both LLM-based optimization and the CoV verification framework is not limited to vectorization. In fact, our results demonstrate that LLMs are capable of performing context-aware optimizations and that CoV can be used to validate such optimizations beyond loop vectorization.

For example, in the case of Figure 9, LLVM -O3 fails to vectorize the original code due to a loop-carried dependency, where the inner loop, governed by the induction variable `j`, depends on the result of the previous iteration through `aa[j-1][i]` (Line 5). Vectorization becomes enabled after separating the dependency-free computation (Line 3) from the data-dependent computation through loop fission and interchanging the loop order, such that the outer loop iterates over `j` and the inner loop over `i` (Lines 10-12). These transformations isolate the dependency and align the data flow with the vector dimension. While such restructuring is difficult for traditional compilers to infer, an LLM can recognize these patterns and apply the required sequence of transformations, enabling vectorization that would otherwise be unreachable.

In summary, CoV illustrates how LLM-based code transformations can complement traditional compiler optimizations by uncovering vectorization opportunities that standard compilers overlook. Beyond loop restructuring such as fission and interchange, LLMs can recognize and apply a wide range of context-sensitive optimizations. Their ability to interpret program semantics at a higher level enables transformations that extend beyond syntactic patterns, motivating the exploration of novel optimization strategies such as memory layout, parallelism, and algorithmic simplification. Pursuing these

broader opportunities also requires addressing practical challenges, including developing more effective prompt-engineering techniques to reliably guide LLM behavior and mitigating token-scale limitations when optimizing large or complex code regions. Nevertheless, overcoming these challenges would further strengthen a promising future for the role of LLMs as intelligent agents within the compiler optimization pipeline, extending beyond loop vectorization for future work.

VIII. Related Work

A. LLMs for Compilers

Recent research [12]–[14], [37] explores compiler designs that leverage LLMs and ML models for performance-critical code transformations, such as loop vectorization, code refactoring, and parallel execution planning. These approaches position language models as co-optimizers within the compilation pipeline, unlocking opportunities that traditional compilers often miss. LLM-Vectorizer [12] introduces a multi-agent system that uses LLMs with feedback loops to generate SIMD-intrinsic-based vectorized code, achieving notable speedups while validating transformations using Alive2. VecTrans [13] employs LLMs to refactor non-vectorizable loops into patterns amenable to compiler auto-vectorization, using IR-level checks to ensure semantic preservation. LLM Compiler [14] develops foundation models trained on low-level code to predict effective compiler optimization options that reduce code size. NeuroVectorizer [37] utilizes deep reinforcement learning within LLVM to learn optimal vectorization strategies, demonstrating the feasibility of ML-driven loop optimization. These works demonstrate that LLMs and ML models can enhance traditional compiler optimization in areas such as vectorization through direct code generation, intelligent code transformation, and optimization decision guidance.

B. Code Verification

Static Verification: Recent approaches [12], [13] integrate static verification tools to check the correctness of LLM-based transformations. Alive2 [17] compares the semantic equivalence between original and transformed LLVM IR by encoding them into logical constraints and using an SMT solver to check whether the outputs match for all possible inputs. Although Alive2 is effective in many cases, its scalability is limited by IR complexity, which may lead to verification failures due to timeouts. Furthermore, Alive2's applicability is constrained when handling loops, requiring additional loop unrolling or transformation to convert the code into a verifiable form.

KLEE [31] detects program differences by systematically generating test inputs that trigger different execution paths. While KLEE enables dynamic exploration beyond static proofs, it suffers from exponential path growth, which limits scalability under diverse branching behavior. These limitations illustrate the challenges of applying formal verification to LLM-optimized code and motivate additional runtime techniques that guarantee correctness where static analysis fails.

Runtime Verification: Several frameworks [38]–[40] have targeted speculative loop parallelization based on the assumption of limited loop-carried dependencies. Du et al. [38] propose a cost-driven compilation framework where speculative threads execute parallel loop iterations, while a main thread monitors memory accesses to detect conflicts and triggers re-execution when conflicts occur. SMTX [39] handles irregular or unresolved dependencies by validating speculative loads and stores at commit time and restoring state when conflicts are detected. DOMORE [40] exploits cross-invocation parallelism by detecting memory dependencies at runtime and stalling threads to avoid violations without re-execution.

Runtime verification has also been applied to speculative vectorization [41]–[45] when static analysis fails due to complex control flow or ambiguous memory access patterns. FlexVec [41] adapts SIMD execution by dynamically adjusting vector lengths and using predicate masks to identify and isolate lanes that violate memory dependencies. Sujon et al. [42] and VecRC [43] speculatively vectorize predicted branches with runtime checks for control-flow divergence, falling back to scalar execution for the affected lanes when speculation fails. SRV [45] introduces hardware support for speculative SIMD vectorization, selectively replaying only the lanes that read incorrect data rather than performing complete rollbacks.

Verifications in speculative execution schemes primarily check for memory or control-flow violations to ensure correctness under their respective optimization schemes. However, when using LLMs as the source of optimization, the resulting strategies are often opaque and unpredictable, making correctness verification more challenging. To address this challenge, CoV adopts a conservative approach that compares all memory stores during execution and falls back to the original code upon detecting any conflicts.

IX. Conclusion

This work presents CoV, a compiler–runtime co-operative Chain of Verification framework that safely integrates LLM-based optimizations into existing compiler pipelines. By combining static checks with runtime verification, CoV broadens the scope of trustworthy LLM-based optimizations. Across the TSVC benchmarks and realistic applications, CoV improves vectorization coverage and delivers performance gains over -O3 while preserving semantic correctness. These results highlight the potential of CoV to safely integrate LLM-based optimizations into traditional compilation pipelines.

Acknowledgment

We thank the CoreLab members for their support and feedback during this work. We also thank the anonymous reviewers and the shepherd for their insightful comments and suggestions. This work is supported by No. RS-2025-02214497, No. RS-2024-00358765, No. RS-2023-00277060, and No. RS-2024-00395134 funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. *(Corresponding author: Hanjun Kim)*

REFERENCES

[1] DeepSeek-AI and D. G. et al., "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," *arXiv preprint arXiv:2501.12948*, 2025.

[2] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: A Family of Highly Capable Multimodal Models," *arXiv preprint arXiv:2312.11805*, 2023.

[3] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving Language Understanding by Generative Pre-Training," 2018. [Online]. Available: https://openai.com/index/language-unsupervised/

[4] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," *arXiv preprint arXiv:2302.13971*, 2023.

[5] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[6] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov *et al.*, "Code Llama: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.

[7] Anthropic, "The Claude 3 Model Family: Opus, Sonnet, Haiku," 2024. [Online]. Available: https://www.anthropic.com/claude-3-model-card

[8] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[9] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman, "CodeGemma: Open Code Models Based on Gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[10] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "StarCoder 2 and The Stack v2: The Next Generation," 2024.

[11] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: empowering code generation with OSS-INSTRUCT," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[12] J. Taneja, A. Laird, C. Yan, M. Musuvathi, and S. K. Lahiri, "LLM-Vectorizer: LLM-Based Verified Loop Vectorizer," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. Association for Computing Machinery, 2025.

[13] Z. Zheng, L. Cheng, L. Li, R. C. O. Rocha, T. Liu, W. Wei, X. Zhang, and Y. Gao, "VecTrans: LLM Transformation Framework for Better Auto-vectorization on High-performance CPU," *arXiv preprint arXiv:2503.19449*, 2025.

[14] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "LLM Compiler: Foundation Language Models for Compiler Optimization," in *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC)*. Association for Computing Machinery, 2025.

[15] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large Language Models for Compiler Optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[16] D. Grubisic, C. Cummins, V. Seeker, and H. Leather, "Compiler generated feedback for Large Language Models," *arXiv preprint arXiv:2403.14714*, 2024.

[17] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for LLVM," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2021.

[18] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO)*, 2004.

[19] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An Evaluation of Vectorizing Compilers," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[20] F. Black and M. Scholes, "The Pricing of Options and Corporate Liabilities," *Journal of Political Economy*, 1973. [Online]. Available: http://www.jstor.org/stable/1831029

[21] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 1979. [Online]. Available: http://www.jstor.org/stable/2346830

[22] X. He and L.-S. Luo, "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation," *Phys. Rev. E*, Dec 1997. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.56.6811

[23] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," *arXiv preprint arXiv:2101.04808*, 2021.

[24] Z. Wang and M. O'Boyle, "Machine Learning in Compiler Optimisation," *arXiv preprint arXiv:1805.03441*, 2018.

[25] H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," in *2020 Forum for Specification and Design Languages (FDL)*, 2020.

[26] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather, "ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations," in *Proceedings of the 38th International Conference on Machine Learning*, ser. ICML'21, 2021.

[27] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "CompilerGym: robust, performant compiler optimization environments for AI research," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022.

[28] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. Leather, and Y. Tian, "Learning compiler pass orders using coreset and normalized value prediction," in *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[29] V. Seeker, C. Cummins, M. Cole, B. Franke, K. Hazelwood, and H. Leather, "Revealing Compiler Heuristics through Automated Discovery and Optimization," in *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024.

[30] P. M. Phothilimthana, A. Sabne, N. Sarda, K. S. Murthy, Y. Zhou, C. Angermueller, M. Burrows, S. Roy, K. Mandke, R. Farahani, Y. E. Wang, B. Ilbeyi, B. Hechtman, B. Roune, S. Wang, Y. Xu, and S. J. Kaufman, "A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers," in *Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2024.

[31] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, 2008.

[32] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, 2008.

[33] Free Software Foundation, "GNU Compiler Collection (GCC)," https://gcc.gnu.org/.

[34] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[36] Standard Performance Evaluation Corporation, "SPEC CPU 2017 Benchmark Suite," https://www.spec.org/cpu2017/, 2017.

[37] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*. Association for Computing Machinery, 2020.

[38] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, "A cost-driven compilation framework for speculative parallelization of sequential programs," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2004.

[39] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," *SIGPLAN Not.*, 2010.

[40] J. Huang, T. B. Jablin, S. R. Beard, N. P. Johnson, and D. I. August, "Automatically exploiting cross-invocation parallelism using runtime information," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[41] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, "FlexVec: auto-vectorization for irregular loops," *SIGPLAN Not.*, 2016.

[42] M. H. Sujon, R. C. Whaley, and Q. Yi, "Vectorization past dependent branches through speculation," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[43] B. Liu, A. Laird, W. H. Tsang, B. Mahjour, and M. M. Dehnavi, "Combining Run-Time Checks and Compile-Time Analysis to Improve Control Flow Auto-Vectorization," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Association for Computing Machinery, 2023.

[44] R. Kumar, A. Martínez, and A. González, "Assisting Static Compiler Vectorization with a Speculative Dynamic Vectorizer in an HW/SW Codesigned Environment," *ACM Trans. Comput. Syst.*, 2016.

[45] P. Sun, G. Gabrielli, and T. M. Jones, "Speculative Vectorisation with Selective Replay," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.