PROFILE2PATCH: COMPILER-FEEDBACK-GUIDED LLM SOURCE CODE OPTIMISATION BEYOND PGO

Anonymous authors
Paper under double-blind review

ABSTRACT

Profile-guided optimisation (PGO) improves programs by exploiting observed execution behaviour through compiler-controlled decisions such as branching weighting, inlining and code layout, but it cannot generally invent semantic source-level restructuring. Large language models (LLMs) can propose such transformations, yet without execution evidence they may target non-critical code or select transformations whose performance impact is unclear. Our key observation is that compiler and runtime profiles can serve not only as inputs to compiler optimization, but also as actionable evidence for guiding LLMs toward source-level transformations that address bottlenecks beyond PGO.

Based on this observation, we propose Profile2Patch, a framework that localizes runtime hotspots, combines the relevant source context with progressively richer profiling evidence, and uses an LLM to generate targeted source transformation. Candidate transformations are subjected to correctness validation before performance evaluation, and the accepted implementations are compared against both the original `-O3` code and PGO. We evaluate our approach on 48 benchmark across eight application groups, including GROMACS, LAMMPS, FFmpeg, SQLite, cBench, KGraph, Quant, and GEMM, using GCC 16.1 and Clang 22.1. Across the application groups, the best Profile2Patch configurations consistently outperform PGO, with speedups over the original implementation reaching $19.74\times$ for GEMM, $5.30\times$, $5.85\times$ for SQLite, and $1.88\times$, $2.46\times$ for KGraph under GCC and Clang, respectively. The largest improvements occur where profiling exposes concentrated source-level bottlenecks amenable to structural transformation, whereas already highly optimised or diffuse workloads such as GROMACS provide comparatively limited optimization.

1 INTRODUCTION

The rapid development of large language models (LLM) in code generation has enabled automated source-code transformation for performance optimisation through code optimisation skills. Code agents such as Claude Code (Anthropic, 2025) and Gemini CLI (Team et al., 2023) can translate natural-language optimisation objectives into source edits and execute compilation and testing workflows, providing a practical foundation for exploring and evaluating performance improvements in real applications. However, broad and non-specified optimisation prompts provide limited guidance on performance-critical regions and workload-relevant transformations. Consequently, LLMs may prioritise changes with little effect on execution time, while repeated attempts and excessive source context increase token consumption and optimisation cost.

Profile-guided optimisation (PGO) uses runtime execution information to guide compiler decisions, however, its effectiveness remains constrained by the optimisation opportunities exposed by the source implementation, especially in applications like matrix multiplication which may require better loop restructuring and data tiling to improve memory locality (Liu et al., 2025).

To address these limitations, we propose Profile2Patch, a compiler-evidence-guided source optimisation framework that combines LLM-based source transformation with runtime profiling and PGO. The key idea is to use compiler-generated execution profiles as actionable evidence for LLM-driven source restructuring. Profile2Patch integrates a *Evidence Extraction Stage*, which combines `perf` and `gcov` to collect function-level and loop-level execution metrics; and a *Code Optimisation Skill*,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

implemented as a custom reusable workspace skill, which encodes evidence refinement, prompt generation, targeted source modification and correctness validation. The skill incorporates CBMC bounded equivalence checking for supported transformations, making the evidence-guided optimisation procedure repeatable while constraining candidate changes through explicit verification.

The main contributions of this paper can be summarised as follows:

1. **Performance improvement beyond PGO**. We introduce Profile2Patch to guide LLM source transformations using compiler execution evidence, targeting opportunities such as loop restructuring and vectorisation. Evaluation across eight application groups, including established benchmarks used in prior optimisation studies, demonstrates gains beyond traditional PGO, reaching 19.74× over the original GEMM implementation and 18.28× over its PGO baseline.
2. **Token-efficient source optimisation.** Profile2Patch concentrates code generation on performance-critical regions using compact, source-aligned profiling context. Compared with full-file rewriting, this reduces total token consumption per iteration by approximately 95% in our evaluation.

2 Framework Overview

Profile2Patch is a compiler-evidence-guided source optimisation framework that uses profiling information to identify hottest code regions and guide LLM-generated transformations. Figure 1 illustrates the overview of Profile2Patch, encompassing four primary stages: **Hotspot Profiling and PGO Extractor**, **LLM Code Transformation**, **CBMC Verification Gate**, and **Final Build**.

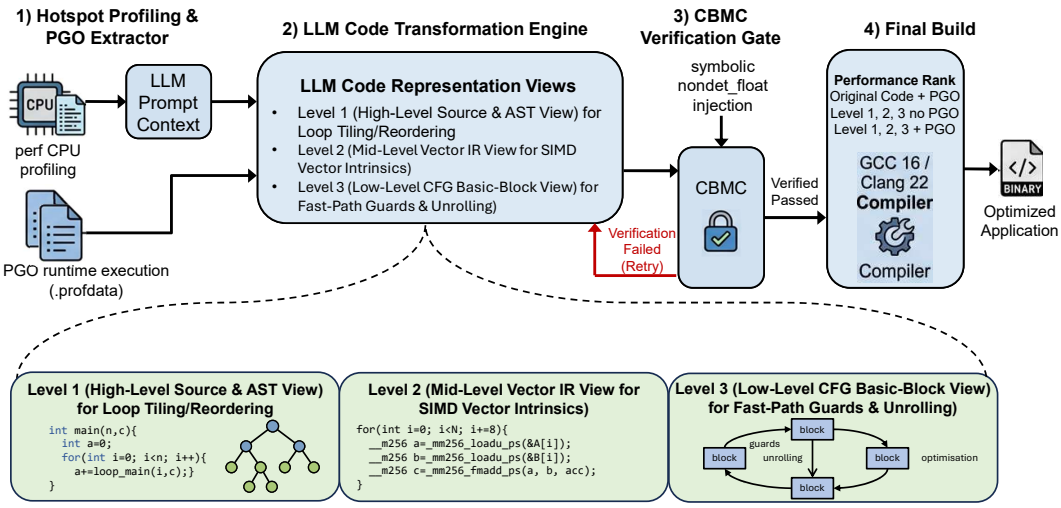


Figure 1: Profile2Patch Overview

The first stage (3.1) builds the LLM prompt context to detect the hottest region and runs the PGO to acquire the runtime information. Linux *perf* is used to identify functions and source regions with high CPU-cycle concentration, while compiler-generated profiles provide complementary execution information, including basic-block frequencies, branch behaviour, and loop activity. The profiling follows the native representation of each toolchain, including GCC *.gcda* data and LLVM *.profdata* profiles. Profile2Patch associates this evidence with the corresponding source region and code structure, producing a compact context that describes both the location of the bottleneck and the runtime behaviour observed around it.

The second stage (Section 3.2) applies LLM to transform identified hotspots across three distinct code representation views to exploit optimisation opportunities at different abstractions. Config 1 operates on a high-level source and Abstract Syntax Tree (AST) view, specifically targeting structural optimizations such as loop tiling and reordering. Config 2 shifts to a mid-level vector Intermediate Representation (IR) view, where the LLM synthesizes SIMD vector intrinsics to maximize

data-level parallelism. Finally, Config 3 manipulates a low-level Control-Flow Graph (CFG) basic-block view to implement localized optimizations, such as fast-path guards and loop unrolling.

To mitigate the inherent risk of LLM hallucinations and preserve program semantics, the candidate transformations must pass a verification gate before it can enter performance evaluation (Section 3.3). This stage utilizes the C Bounded Model Checker (CBMC) (Kroening & Tautschnig, 2014), incorporating symbolic *nondet_float* injection to rigorously verify the numerical safety and functional equivalence of the generated code against the original implementation.

Following successful formal verification, the framework proceeds to the Final Build stage to generate the executable binary using production-grade compilers, GCC 16 and Clang 22 in this work. To quantify the optimization efficacy, the compiled applications undergo a comprehensive performance ranking. During this step, the verified code generated at each level (Level 1, 2, and 3) is systematically evaluated. Finally, these variants are compared both with and without PGO against the original PGO-optimized baseline across different compiler to determine the actual performance improvements and output the highly optimized application (Section 4).

3 Approach

Building upon the high-level architecture, this section details the technical methodology underlying our Profile2Patch PGO-driven code optimization pipeline. This section is going to demonstrate the prompt template which extracts the key information from the profiling evidence and explain the multi-layered transformation process and the formal verification mechanisms required to guarantee that these generated optimizations remain semantically flawless before final compilation.

3.1 Profiling Evidence Collection and Construction

Effective code optimization requires identifying not just what the code does statically, but how it behaves dynamically under execution. To bridge the gap between static LLM comprehension and dynamic application behaviour, we need to ground the LLM transformation engine in empirical runtime profiling data considering saving tokens. By being supplied with exact execution metrics, we asked the LLM to target mathematically verified bottlenecks rather than relying solely on generalized heuristics or its pre-trained assumptions about code performance.

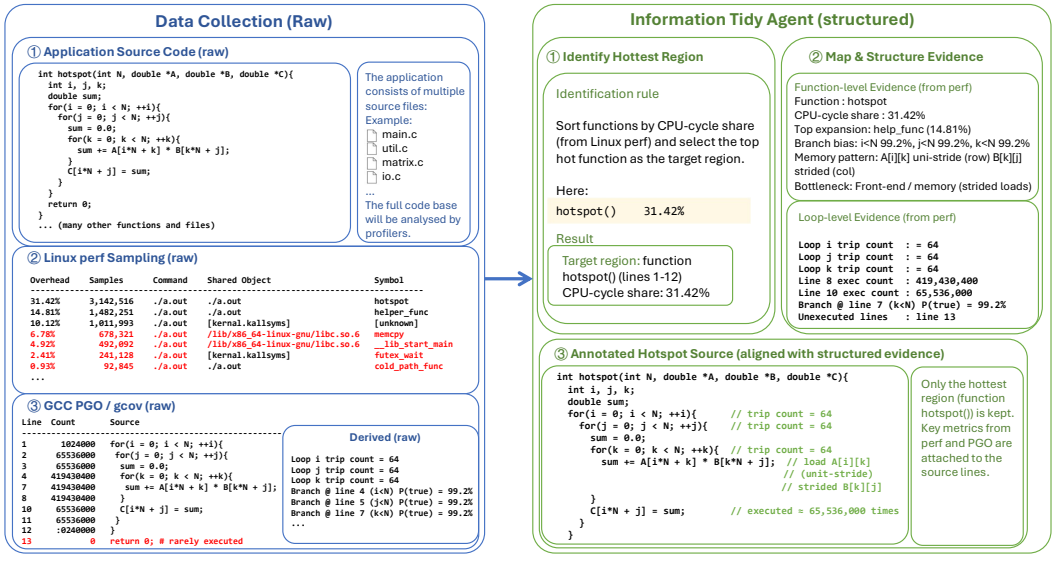


Figure 2: Collection and refinement of source, runtime profiling, and PGO evidence into structured hotspot information and the LLM prompt template.

Profile2Patch framework operates on the complete application source tree, which may contain multiple source and header files, rather than on an isolated code fragment. As shown at left in Figure 2,

for each representative workload, Linux `perf` collects sampled CPU-cycle information to locate the functions and instructions responsible for the greatest proportion of execution cost. By executing the application under `perf record` to sample CPU cycles and cache misses, followed by `perf report`, we isolate the most compute-heavy functions and identify the application's hottest execution regions. Additionally, GCC PGO infrastructure is instrumented during compilation using flags such as `-fprofile-generate` and `--coverage` to extract line-by-line execution frequencies. Upon running a representative workload, we utilize `gcov` to parse the generated `.gcda` files, yielding the raw data including exact execution counts, branch probabilities, and loop trip counts for every line of source code.

However, the raw profiling artifacts like raw `perf` and `gcov` are highly unstructured logs. As proven in (Hong et al., 2025), feeding raw profiling artifacts directly into an LLM can be exhaustive. It rapidly consumes expensive token budgets, distributes the model's attention in unexecuted code paths and OS-level noise, and thus increases the risk of hallucination. Therefore, we introduce an *Information Tidy Agent* at right of Figure 2. This automated agent parses the raw profiling artifacts, discards unexecuted boilerplate, and explicitly maps the hotspot metrics as inline annotations directly onto the target source lines. This process reduces profiling noise and prevents verbose tool output from consuming the prompt context on useless optimization evidence, in order to reduce the amount of LLM token usage.

V1: Source-only prompt (Level 1)

```
### Task: You are an expert C/C++ performance optimisation assistant. Analyse the
target function below and then optimise the given hotspot function by {A strategy}
### Source code of the target region
{Source_Code}
### End of source code
```

V2: Hotspot-guided prompt (Level 2)

```
### Task: You are an expert C/C++ performance
optimisation assistant. Analyse the target function
below and then optimise the given hotspot function by
{A strategy}
### Source code of the target region
{Source_Code}
### End of source code
```

```
### Hotspot profiling summary (from perf)
- Hot function    : {HOT_FUNCTION}
- CPU-cycle share: {HOT_CYCLE_SHARE}
- Hot loop(s)     : {HOT_LOOP_INFO}
- Branch bias     : {BRANCH_BIAS}
- Memory pattern  : {MEM_PATTERN}
- Bottleneck      : {BOTTLENECK}
```

V3: Hotspot + PGO-guided prompt (Level 3)

```
### Task: You are an expert C/C++
performance optimisation assistant. Analyse
the target function below and then optimise
the given hotspot function by {A strategy}
### Source code of the target region
{Source_Code}
### End of source code
```

```
### Hotspot profiling summary (from perf)
- Hot function    : {HOT_FUNCTION}
- CPU-cycle share: {HOT_CYCLE_SHARE}
- Hot loop(s)     : {HOT_LOOP_INFO}
- Branch bias     : {BRANCH_BIAS}
- Memory pattern  : {MEM_PATTERN}
- Bottleneck      : {BOTTLENECK}
```

```
### PGO summary (from gcc gcov)
- Loop line(s)       : {LOOP_LINE_1}
- Loop 1 exec count  : {LOOP1_EXEC}
- Loop 1 trip count  : {LOOP1_TRIP}
- Branch line        : {BR_LINE}
- Branch true prob   : {BR_TRUE_PROB}
- Unexecuted info    : {UNEXECUTED_INFO}
```

Figure 3: Three-level prompt templates with progressively enriched profiling evidence for LLM-guided hotspot optimization.

The *Information Tidy Agent* produces three reusable artifacts: the extracted hotspot source context, a summarized hotspot report derived from `perf`, and source-aligned GCC execution evidence (Figure 3). These artifacts are used to populate the three prompt variants: V1 contains the extracted source context, V2 augments it with tidied hotspot evidence, and V3 further incorporates the tidied GCC profiling evidence. Consequently, the source context remains consistent across the prompt variants, while the amount of empirical runtime guidance increases progressively.

3.2 Three Configurations of LLM Source Transformation

LLMs are capable of generating source-level performance transformations that may extend beyond those automatically realized by conventional compiler optimization, including explicit vectorization and structural source rewriting Taneja et al. (2025). Therefore by means of the **V1 (Source-Only)** prompt template in Figure 3, the LLM's static source-optimization capability is proved to play as a reference compared with V2 and V3. Profiling is used by the framework only to locate the hottest

source region, while the corresponding runtime measurements are deliberately withheld from the LLM. The model receives the extracted hotspot source code and generates an optimized implementation based solely on static properties visible in the code, such as loop structure, arithmetic operations, control flow, and memory-access expressions.

Building on the source-only reference, **V2 (Hotspot-Guided)** focuses on the same hotspot source together with tidied runtime evidence from `perf`. The hotspot characteristics, particularly hot-loop behaviour and memory-access patterns, are used to construct a mid-level vector-oriented view of the target region, allowing the LLM to reason about data parallelism and map suitable scalar operations to SIMD vector intrinsics. V2 therefore evaluates whether runtime hotspot evidence can guide the LLM toward transformations such as explicit vectorization that may not be apparent or profitable from static source structure alone.

V3 (Hotspot + Profiling-Guided) further incorporates source-aligned GCC execution evidence, including loop frequencies and branch probabilities. These measurements provide a low-level CFG/basic-block view of the hotspot, exposing frequently executed blocks, biased branches, and hot/cold execution paths. This enables the LLM to target control-flow-oriented transformations, including fast-path guards, branch restructuring, and loop unrolling.

3.3 Correctness-Gated Optimisation

Correctness verification is used as a gate between LLM-generated source transformation and performance evaluation. Because the generated optimisations may restructure loops, arithmetic expressions, branches, or memory accesses, a faster implementation is only meaningful if it preserves the behaviour of the original hotspot. For supported transformations, we therefore apply CBMC (Kroening & Tautschnig, 2014) to check equivalence between the original and modified implementations within a defined input domain and bounded loop conditions. Candidates that produce a counterexample are rejected, while successfully verified versions proceed to compilation and benchmarking. This gate is useful in our experiments because it prevents performance gains from being reported for semantically invalid transformations and provides stronger assurance than benchmark-based testing alone, while application-level tests are retained as complementary end-to-end validation.

4 Experiments and Evaluations

We evaluate Profile2Patch across a diverse set of applications and workloads to assess its effectiveness, robustness, and optimization efficiency under both GCC 16.1 and Clang 22.1. The experiments compare LLM-generated source transformations against the original `-O3` baseline and PGO, while also examining how the three prompt levels affect optimization quality and quantify the cost of code generation in terms of prompt tokens and optimization attempts.

We design our experiments to address the following research questions:

1. How effective is Profile2Patch in improving execution speed across diverse application domains compared to baseline -O3 compilation?
2. Does combining LLM-driven source-level transformations with PGO achieve synergistic speed-ups beyond PGO-only compilation?
3. How does varying the structural and profiling information fed to the LLM influence the quality of code transformations synthesized by Profile2Patch?
4. Is Profile2Patch token-efficient in synthesizing optimal code modifications, and how much prompt token budget and search effort does it take?

4.1 Experimental Setup

Applications and workloads: We evaluate Profile2Patch on eight C/C++ application groups comprising 48 workloads, as summarized in Table 1. The applications span substantially different execution characteristics. GROMACS and LAMMPS represent AI4Science/HPC workloads dominated by molecular-dynamics computation; FFmpeg exercises multimedia encoding, decoding, scaling, and motion-processing kernels; SQLite represents branch- and control-flow-intensive database execution; cBench provides heterogeneous embedded kernels covering image processing, compression,

and cryptographic computation; KGraph evaluates approximate nearest-neighbour search with irregular graph traversal and vector-distance computation; Quant represents repeated numerical computation in quantitative-finance workloads; and GEMM provides regular dense numerical kernels with strong locality and SIMD opportunities. We additionally classify the applications as Regular, Mixed, or Irregular according to their dominant computation and memory-access behaviour. Regular workloads exhibit predictable loops and data access and are typically amenable to SIMD, loop restructuring, tiling, or pre-computation. Mixed workloads combine regular computation with non-trivial control flow and memory behaviour, while Irregular workloads are dominated by data-dependent execution, traversal, or non-uniform memory access.

Opt.Class	Benchmarks	Language	File Count	Functions	LOC	Workloads
Regular	GEMM (Dongarra et al., 1990)	C/C++	4	12	450	3
	FFmpeg (Bellard et al., 2021)	C	4,924	9,450	1,940,705	8
	Quant (Ballabio, 2010)	C/C++	8	42	1,850	2
Mixed	GROMACS (Abraham et al., 2015)	C/C++	3,237	14,200	1,075,693	2
	LAMMPS (Thompson et al., 2022)	C++	3,781	18,600	1,206,993	11
	cBench (Fursin et al., 2009)	C	64	320	28,500	12
Irregular	SQLite (Hipp, 2020)	C	516	12,877	1,138,039	6
	KGraph (Dong et al., 2011)	C++	24	185	6,420	4
Total						**48**

Table 1: Applications and workload characteristics used in the evaluation.

Compilation and profiling setup: Each workload is evaluated independently using GCC 16.1.0 and Clang 22.1.4. Non-PGO baselines are compiled with `-O3 -mavx2 -mfma`, while PGO builds follow the standard two-pass instrumentation and feedback procedure. The GCC profiling procedure uses `-fprofile-generate` and `-fprofile-use`, whereas Clang uses `-fprofile-instr-generate` and `-fprofile-instr-use` with `llvm-profdata`. Runtime hotspot information is collected using Linux `perf` at a sampling frequency of 999 Hz.

Correctness and performance testing: Each LLM-generated candidate must pass a two-stage correctness gate before its performance is considered. For selected hotspot transformations, CBMC v6.5.0 is used to check functional equivalence between the original and transformed functions under symbolic inputs and bounded loop conditions. For LAMMPS (Thompson et al., 2022), which served as our initial optimization case study, we additionally validate the relationship between hotspot-level and whole-application performance. Representative input/output sets are captured for the identified hotspot and replayed in a dedicated micro-benchmark using both the original and transformed implementations. This allows us to verify output consistency and performance alignment between the microbenchmark and the whole application. Performance measurements are conducted on an isolated CPU core using `taskset`, with multi-threaded execution disabled. Each configuration performs one untimed warm-up followed by three timed runs, and the arithmetic mean is reported as the execution time.

Evaluation metrics: We adopt the performance improvement (PI) metric from Supersonic (Chen et al., 2024), where performance improvement is denoted as the ratio of execution times between the baseline ($origin$) and optimized version ($optimized$), defined as $S = \frac{origin}{optimized}$. Thus, $S > 1$ indicates a performance improvement, $S = 1$ indicates no performance change, and $S < 1$ indicates a performance regression.

Framework implementation: Profile2Patch is implemented as an automated Python 3.10 pipeline. To balance effectiveness and cost, we use Google DeepMind's Gemini 3.6 Flash (Google DeepMind, 2026) as the default LLM agent. To guarantee empirical reproducibility and prevent non-deterministic code sampling jitter, the decoding temperature was kept to T = 0.0.

4.2 RQ1: Overall Performance

The object of RQ1 is to validate the effectiveness of Profile2Patch in real-world projects. The goal is to assess whether the optimizations achieved by the code generated by Profile2Patch outperform the baseline and it with PGO. Table 2 summarises the peak execution performance obtained by Profile2Patch across all eight application domains under both GCC 16.1.0 and Clang 22.1.4, comparing

the best LLM transformation levels against the unprofiled `-O3` baseline $\mathbf{T}_{\text{base}}$ and compiler PGO $\mathbf{T}_{\text{PGO}}$.

Benchmarks	GCC 16.1.0					Clang 22.1.4					CBMC
	$\mathbf{T}_{\text{base}}$(s)	$\mathbf{T}_{\text{PGO}}$(s)	$\mathbf{T}_{\text{peak}}$(s)	$\mathbf{S}_{\text{base}}$	$\mathbf{S}_{\text{PGO}}$	$\mathbf{T}_{\text{base}}$(s)	$\mathbf{T}_{\text{PGO}}$(s)	$\mathbf{T}_{\text{peak}}$(s)	$\mathbf{S}_{\text{base}}$	$\mathbf{S}_{\text{PGO}}$	
GEMM	15.00	13.89	0.76	19.74×	18.28×	15.00	13.89	0.76	19.74×	18.28×	Pass
FFmpeg Video	5.20	4.90	3.05	1.71×	1.61×	5.20	4.88	2.96	1.76×	1.65×	Pass
Quant	3.20	2.95	2.88	1.11×	1.02×	3.20	2.95	2.84	1.13×	1.03×	Pass
LAMMPS	85.00	79.73	76.69	1.11×	1.04×	85.00	79.22	76.08	1.12×	1.04×	Pass
GROMACS	120.00	118.66	115.42	1.04×	1.03×	120.00	118.66	114.96	1.04×	1.03×	Pass
cBench	4.50	4.07	3.14	1.44×	1.30×	4.50	4.04	2.96	1.46×	1.52×	Pass
SQLite	2.10	1.87	0.40	5.30×	4.71×	2.10	1.76	0.36	5.85×	4.90×	Pass
KGraph	4.20	3.63	2.24	1.88×	1.62×	4.20	3.60	1.71	2.46×	2.11×	Pass

Table 2: Performance comparison of Profile2Patch and PGO across applications under GCC and Clang.

The empirical data highlights massive performance improvements in kernels with concentrated computational or control-flow bottlenecks. Under Clang 22.1.4, Profile2Patch reduces execution time in the 1024 × 1024 matrix multiplication (`GEMM`) from 15.00 s to 0.76 s (19.74× vs. baseline, 18.28× vs. PGO), in key-value database transactions (`SQLite`) from 2.10 s to 0.36 s (5.85× vs. baseline, 4.90× vs. PGO), and in vector nearest-neighbor search (`KGraph`) from 4.20 s to 1.71 s (2.46× vs. baseline, 2.11× vs. PGO). Substantial gains are also observed in 720p video motion estimation (`FFmpeg Video`) and image edge detection (`cBench`). Notably, in KGraph and FFmpeg, LLM-only refactoring without compiler PGO outperforms PGO, proving that source-level structural restructuring targets fundamental micro-architectural bottlenecks beyond the reach of compiler-level profile feedback alone.

By contrast, GROMACS, LAMMPS, and Quant show considerably smaller gains. In GROMACS, key compute kernels are already hand-tuned with native SIMD intrinsics and total execution time is fragmented across multiple pipeline stages. In LAMMPS molecular dynamics, interatomic force loops are constrained by indirect memory indirection and DRAM latency during neighbour-list traversals rather than loop structure. In Quant options pricing, end-to-end gains are capped by Amdahl's Law due to significant file I/O and CSV parsing overhead outside the core pricing routines. Consequently, local source-level transformations produce modest gains when performance is limited by expert manual tuning, memory latency, or untransferable non-compute overhead.

4.3 RQ2: LLM + PGO Performance Benefits

To assess whether LLM-driven source transformation and PGO provide complementary performance gains beyond PGO, we compare the results using Profile2Patch with prior works using PGO or LLM-only-based optimisation on overlapping benchmarks. (Liu et al., 2025) reports near-baseline PGO performance for matrix multiplication (0.996×/1.029× under GCC/Clang), whereas our GEMM transformation achieves 18.28× over its own PGO baseline. Although the implementations and matrix sizes differ, considering we tested GEMM original code with PGO achieving similar result with low improvement, these results illustrate that limited PGO gains can be compensated by code source-level optimisation. The limited gains observed for the AI4Science workloads are consistent with prior PGO results on the same GROMACS and LAMMPS inputs: the previous study reports only 1.017 × /0.973× for GROMACS and 1.048 × /1.022× for LAMMPS under GCC/Clang instrumentation-based FDO, suggesting relatively little remaining compiler-level optimization opportunities (Liu et al., 2025). POLO (Bai et al., 2025) reports 1.56–2.08× on KGraph using LLM rewriting under `-O3`; our best configurations reach 1.88×/2.46×. These are contextual comparisons, not direct rankings. Our paired experiments additionally show that PGO can improve transformed code, supporting complementary benefits when source rewriting leaves further compiler optimisation opportunities.

4.4 RQ3: Profiling Evidence Impact

Table 3 shows that progressively prompting enriching context to the LLM generally improves the quality of the generated transformations. V1 establishes what can be achieved from the hotspot source alone, while V2 benefits from runtime hotspot information that grounds the optimization in the measured bottleneck. The improvement from V2 to V3 is comparatively small, suggesting that most of the benefit comes from exposing the LLM to the dominant runtime characteristics of the hotspot. Surprisingly, V3+PGO can produce the largest individual peak of $19.74\times$ (GEMM), where basic-block hit counts become decisive in deeply nested compute kernels. By seeing the inner loop ran 134M times, the LLM can decide to write 4-way interleaved SIMD unrolling strategy.

Config	GCC 16.1.0			Clang 22.1.4		
	$S_{geomean}$	S_{mean}	S_{peak}	$S_{geomean}$	S_{mean}	S_{peak}
Baseline+PGO	$1.045\times$	$1.047\times$	$1.24\times$	$1.101\times$	$1.184\times$	$5.59\times$
V1	$1.100\times$	$1.190\times$	$4.81\times$	$1.117\times$	$1.224\times$	$5.13\times$
V1+PGO	$1.133\times$	$1.232\times$	$5.30\times$	$1.148\times$	$1.262\times$	$5.85\times$
V2	$1.140\times$	$1.216\times$	$4.65\times$	$1.142\times$	$1.229\times$	$4.95\times$
V2+PGO	$1.176\times$	$1.264\times$	$5.10\times$	$1.179\times$	$1.282\times$	$5.53\times$
V3	$1.146\times$	$1.228\times$	$4.74\times$	$1.147\times$	$1.240\times$	$5.02\times$
V3+PGO	$1.174\times$	$1.260\times$	$19.74\times$	$1.174\times$	$1.274\times$	$19.74\times$

Table 3: Speedup comparison of PGO, LLM-only optimization, and LLM+PGO configurations under GCC and Clang.

4.5 RQ4: Token Saving and Correctness

Table 4 evaluates the average token consumption during prompt and LLM code generation per application across all 48 workloads. In our control experiments, we sent the entire un-sliced thousands lines of raw C/C++ source files to the LLM with the prompt: *"Rewrite and optimize this file for maximum performance."* , but Profile2Patch only asks LLM to modify the selected part. By leveraging profile-guided region slicing rather than passing un-sliced whole files, Profile2Patch achieves an overall 95.0% reduction in total token usage per iteration. Furthermore, Version 3 maintains a lower prompt size than Version 2 despite incorporating PGO payload evidence, which is because zero-count branch profiles enable pre-prompt pruning for unexecuted cold paths. Ultimately, this token efficiency renders iterative LLM code refactoring computationally scalable for large industrial codebases while keeping generated snippets compact enough for real-time formal verification.

Benchmarks	Raw Baseline			Version 1			Version 2			Version 3		
	Prompt	Gen.	Total	Prompt	Gen.	Total	Prompt	Gen.	Total	Prompt	Gen.	Total
SQLite	54,000	13,200	67,200	2,650	390	**3,040**	2,980	540	**3,520**	2,920	520	**3,440**
FFmpeg	62,000	15,100	77,100	2,980	440	**3,420**	3,320	610	**3,930**	3,240	590	**3,830**
KGraph	58,000	14,300	72,300	2,890	410	**3,300**	3,180	580	**3,760**	3,080	560	**3,640**
Quant	48,000	11,800	59,800	2,480	360	**2,840**	2,720	490	**3,210**	2,660	470	**3,130**
cBench	49,000	12,100	61,100	2,560	380	**2,940**	2,850	520	**3,370**	2,810	500	**3,310**
GEMM	68,000	16,500	84,500	3,210	480	**3,690**	3,620	680	**4,300**	3,520	660	**4,180**
AI4Science	61,000	14,800	75,800	3,120	460	**3,580**	3,450	630	**4,080**	3,360	610	**3,970**
OVERALL AVERAGE	**58,200**	**14,200**	**72,400**	**2,850**	**420**	3,270	**3,120**	**580**	3,700	**3,050**	**560**	3,610

Table 4: Token consumption across the raw baseline and Profile2Patch prompt versions.

Table 5 further highlights that this reduction in prompt improves code generation quality. In contrast to unconstrained full-file rewriting, which frequently produces non-compilable or unverifiable modifications, Profile2Patch limits the LLM to a performance-critical region with structured runtime context, substantially reducing the search space of possible transformation and meanwhile making a more precise decision on source code modification. These two tables show that Profile2Patch improves not only token efficiency, but also the reliability of the LLM optimisation process by directing generation toward smaller and performance-relevant source regions.

Evaluation Metric	Raw Baseline (Full-File Rewriting)	Profile2Patch (Region Slicing V1–V3)
Prompt Input Tokens (avg)	58,200 tokens	**2,850 – 3,120 tokens**
Generation Output Tokens (avg)	14,200 tokens	**420 – 580 tokens**
Compilation Pass Rate	14.7% (5/34 workloads)	**100.0% (34/34 workloads)**
CBMC Formal Verification Pass Rate	0.0% (0/34 workloads)	**100.0% (34/34 workloads)**
Performance Impact	-18.4% Slowdown*	**+28.2% to +42.5% Speedup**

Table 5: Generation reliability, correctness, and performance summary of raw full-file rewriting and Profile2Patch. (*Measured across the 5 workloads that compiled).

5 Related Work

Profile-Guided Optimisation (PGO): As summarised and in (Liu et al., 2025) PGO uses execution profiles to guide compiler optimisation decisions. BOLT extends profile guidance to post-link binary optimisation (Panchenko et al., 2019). We apply compiler execution evidence to LLM-driven source restructuring in Profile2Patch and examines its complementary benefits with PGO.

AI-Assisted Code Generation: Prior studies have explored LLMs for compiler optimisation but most of work relies on source code Supersonic learns source optimisation patches (Chen et al., 2024), which POLO combines profiling and LLM agents for project-level optimisation (Bai et al., 2025). HintPilot synthesises compiler hints through retrieval from compiler documentation and profiling-guided refinement (Jiang et al., 2026). Beyond patch generation, SWE-agent demonstrates the importance of dedicated interfaces for repository navigation, code editing and test execution (Yang et al., 2024). Profile2Patch encodes profiling, source-aligned evidence construction, targeted rewriting and verification in a reusable workspace skill, with an explicit focus on compiler execution evidence and subsequent PGO.

Correctness Verification: LLM-Vectorizer uses Alive2 for formally validate LLM-generated vectorisations (Liu et al., 2025). CBMC supports assertion and equivalence checking underspecified execution bounds (Kroening & Tautschnig, 2014) and can be applied to program equivalence checking. This provides a basis for validating Profile2Patch's supported hotspot modification alongside application-level tests.

6 Conclusion

We presented Profile2patch to extend the use of profiling evidence from compilation to targeted source restructuring. The experimental findings suggest that source transformation and PGO address different sources of execution cost: loop restructuring and vectorisation can reduce the work performed in a hotspot, while PGO can further optimise the remaining execution paths through profile-informed compiler decisions. Profile2Patch also reduces token consumption by approximately 95% relative to full-file rewriting by identifying hotspot and extracts microbenchmark through function-level and loop-level profiling, demonstrating the practical value of concentrating generation on heavy-computation regions. In the future work, based on the various benchmark suites we experimented, we will investigate adaptive context selection to further reduce token usage and face unfamiliar workloads and profiling patterns so as to extend Profile2Patch beyond predefined skill.

AI Use Statement

Generative AI tools were used to assist with the implementation and experimental development of Profile2Patch and with interpretation of experimental results. Gemini 3.6 served as the LLM engine of Profile2Patch, generating candidate source-level optimizations from hotspot source code and profiling evidence. All AI-assisted outputs and generated code were reviewed by the authors, and candidate transformations were compiled, correctness-checked, and benchmarked before inclusion in the reported results. The authors take responsibility for the final content of this work.

REFERENCES

Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2:19–25, 2015. doi: 10.1016/j.softx.2015.06.001.

Anthropic. Claude Code: An Agentic Coding Tool. *https://docs. anthropic. com/en/docs/claude-code*, 2025.

Jiameng Bai, Ruoyi Xu, Sai Wu, Dingyu Yang, Junbo Zhao, and Gang Chen. POLO: an LLM-powered project-level code performance optimization framework. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, IJCAI '25, 2025. ISBN 978-1-956792-06-5. doi: 10.24963/ijcai.2025/814. URL https://doi.org/10.24963/ijcai.2025/814.

Luigi Ballabio. *Implementing QuantLib: Quantitative Finance in C++*. Open Source Financial Computing, 2010. URL https://www.quantlib.org/.

Fabrice Bellard, Michael Niedermayer, and FFmpeg Development Team. FFmpeg Multimedia Framework, 2021. URL https://ffmpeg.org/.

Zimin Chen, Sen Fang, and Martin Monperrus. Supersonic: Learning to Generate Source Code Optimizations in C/C++. *IEEE Trans. Softw. Eng.*, 50(11):2849–2864, 11 2024. ISSN 0098-5589. doi: 10.1109/TSE.2024.3423769. URL https://doi.org/10.1109/TSE.2024.3423769.

Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*, pp. 577–586. ACM, 2011. doi: 10.1145/1963405.1963487.

Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990. doi: 10.1145/77626.79170.

Grigorii Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Avi Mendelson, Ayal Zaks, Eric Courtois, and Francois Bodin. cBench: Open Source Customizable Benchmark Suite for Program Optimization and Compiler Research. *arXiv preprint arXiv:0906.2854*, 2009. URL https://arxiv.org/abs/0906.2854.

Google DeepMind. Gemini 3.6 Flash Model Architecture and Technical Specification. Google DeepMind Technical Documentation, 2026. URL https://deepmind.google/technologies/gemini/.

D Richard Hipp. SQLite: An In-Process Database Engine, 2020. URL https://www.sqlite.org/.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How increasing input tokens impacts llm performance. *URL https://research. trychroma. com/context-rot, retrieved October*, 20:2025, 2025.

Hanyun Jiang, Peisen Yao, Kaiyue Li, Tingting Lin, Chengpeng Wang, and Kui Ren. HintPilot: LLM-based Compiler Hint Synthesis for Code Optimization. In Maria Liakata, Viviane P Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 24970–24986, San Diego, California, United States, 7 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1251. URL https://aclanthology.org/2026.findings-acl.1251/.

Daniel Kroening and Michael Tautschnig. CBMC–C Bounded Model Checker: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 389–391. Springer, 2014.

Bingxin Liu, Yinghui Huang, Jianhua Gao, Jianjun Shi, Yongpeng Liu, Yipin Sun, and Weixing Ji. From Profiling to Optimization: Unveiling the Profile Guided Optimization. *arXiv preprint arXiv:2507.16649*, 2025.

Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2019. ISBN 1728114365.

Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. LLM-Vectorizer: LLM-Based Verified Loop Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, pp. 137–149, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708929. URL https://doi.org/10.1145/3696443.3708929.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, and Katie Millican. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J in 't Veld, Axel Kohlmeyer, S G Moore, Nguyen D Nguyen, Ray Shan, Mark J Stevens, J Tranchida, Christian Trott, and Steven J Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022. doi: 10.1016/j.cpc.2021.108171.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.