AICORE: COMPILER-GUIDED JOINT SOURCE AND PARAMETER OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based systems for optimizing compiled C/C++ programs typically operate at either the source level or the compiler level. Treating these decisions separately overlooks their coupling: a promising source rewrite can change backend behavior, while compiler-only search cannot introduce transformations that require high-level algorithmic reasoning. Existing compiler interfaces further complicate integration by scattering optimization feedback across low-level diagnostics, intermediate representations, and configuration options. We present AICORE, an agent–compiler co-design framework built around an *AI-native compiler interface*. The interface identifies active code regions, organizes source-linked optimization remarks, pass traces, and IR excerpts into structured reports, and connects this feedback to three action types: source rewrites, loop-local pragmas, and supported compiler parameters. AICORE searches these actions jointly through a shared compile–validate–measure loop. A candidate is retained only if it builds, matches the reference output, and improves performance; after each accepted change, the compiler feedback is refreshed from the new program. On 30 PolyBench/C kernels and 29 cBench programs, AICORE achieves arithmetic-mean speedups of $4.49\times$ and $1.30\times$ over $-O3$, respectively, with a 100.00% validation pass rate on both suites. It exceeds our AutoPass reimplementation, instrumentation-based PGO, and OpenTuner in mean speedup on both suites. OpenCode achieves higher means of $6.33\times$ and $2.20\times$, but passes validation on only 76.67% and 86.21% of the programs. On PolyBench, the full system also exceeds the tested rewrite-only, parameter-only, and no-compiler-evidence variants. Validation establishes output agreement only for the evaluated executions.

1 INTRODUCTION

Code agents such as Codex, Claude Code, and OpenCode are increasingly used to generate programs, implement performance-critical kernels, and automate software-engineering workflows. As their capabilities advance, they are being applied not only to routine programming tasks but also to complex systems work, including the development of prototype compilers and deep-learning frameworks.

One particularly promising application is the optimization of performance-critical C/C++ programs compiled into native binaries. Existing LLM-based approaches broadly operate at two levels: source-level methods rewrite programs to expose more efficient computations, whereas compiler-level methods tune passes, parameters, or other compilation decisions.

The first category uses LLMs to generate or rewrite source code. OpenCode Anomaly Innovations (2025) is a general-purpose open-source coding agent that supports multiple LLM backends. SBLLM combines LLM-generated transformations, evolutionary search, and retrieved optimization patterns to iteratively improve Python and C++ programs Gao et al. (2025). POLO uses profiling and call-graph analysis to identify hotspots and applies LLM-generated optimizations to performance-critical functions Bai et al. (2025). Other systems target specific optimization domains, including GEMM Zhou et al. (2025), loop vectorization Taneja et al. (2025), and CUDA kernel generation Dong et al. (2025).

The second category uses machine-learning models, including LLMs, to guide compiler decisions such as pass ordering and parameter tuning. ECCO learns from records of code transformations

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.

performed by compiler passes, proposes an initial LLVM pass sequence, and refines it through a genetic algorithm Pan et al. (2026). AutoPass Li et al. (2026), the system most closely related to ours, exposes compiler optimization states and intermediate representations to an LLM, enabling it to coordinate pass sequences and parameter settings. Unlike our approach, however, AutoPass does not rewrite source code and relies on compiler interfaces designed primarily for human users. Magellan Chen et al. (2026) operates at a different level, using evolutionary search to generate patches to the compiler's optimization passes themselves.

These two lines of work typically optimize either source code or compiler decisions rather than treating them as a coupled search space. Source-rewriting methods cannot directly control backend mechanisms such as register allocation, whose behavior depends on target-specific registers, live ranges, calling conventions, and register pressure. A seemingly beneficial rewrite may therefore lengthen live ranges, increase register pressure, and introduce spills; its value can be assessed reliably only after observing the resulting compiler behavior.

Conversely, compiler-only approaches are limited to the information and transformations available inside the compiler, so they may miss opportunities that require understanding the program's high-level purpose. For example, a conventional compiler generally cannot replace a hand-written matrix multiplication with an optimized BLAS library routine. Such a replacement requires recognizing the algorithm and determining whether it preserves behavior under the program's assumptions about memory overlap, data layout, numerical computation, and side effects. These complementary limitations motivate combining the high-level reasoning of LLMs with the low-level analysis and optimization capabilities of compilers.

Achieving this integration remains difficult because compiler interfaces are not designed to explain optimization decisions to autonomous agents. For example, when a compiler fails to vectorize a loop, that is, execute several loop iterations in parallel using a single processor instruction, it may emit low-level diagnostics or intermediate code, but the agent must still determine what blocked the transformation. A possible data dependency may make vectorization unsafe, whereas the compiler's cost model may simply predict that vectorization would not be profitable; these two cases require different remedies. More generally, relevant evidence is scattered across optimization remarks, intermediate representations, pass reports, flags, pragmas, and attributes. Turning these signals into a clear decision about whether to rewrite the source code, add a local hint, or adjust a compiler parameter is therefore challenging.

To address these limitations, we present AICORE, an agent–compiler co-design framework that couples a code agent with an *AI-native compiler interface*. The interface makes existing compiler facilities usable by an autonomous agent in three ways. First, it organizes source-linked optimization remarks, pass traces, and IR probes into structured evidence showing which transformations succeeded, failed, or were judged unprofitable. Second, it connects this evidence to concrete actions: source rewrites, loop-local pragmas, and supported compiler parameters. Third, it compiles, validates, and measures every candidate, then refreshes the compiler evidence after each accepted change. This design enables the agent to reason jointly about source code and compiler behavior.

We evaluate AICORE on 30 PolyBench/C kernels and 29 cBench programs. Relative to `-O3`, it achieves arithmetic-mean speedups of $4.49\times$ and $1.30\times$, respectively, with a 100.00% validation pass rate on both suites. It exceeds our AutoPass reimplementation, instrumentation-based PGO, and OpenTuner in mean speedup on both suites. OpenCode achieves higher arithmetic means of $6.33\times$ and $2.20\times$, but passes validation on only 76.67% and 86.21% of the programs. On PolyBench, the full system also exceeds the tested rewrite-only, parameter-only, and no-compiler-evidence variants. These results support the full-system design, but the variants follow different search paths and therefore do not isolate the causal effect of any individual component. All comparisons use the recorded configurations, and validation establishes output agreement only for the evaluated executions.

2 Related Work

LLMs have been used to edit source programs and to guide compiler decisions. One line of work uses execution feedback and search to optimize source programs. SBLLM combines LLM generation with evolutionary search and retrieves optimization patterns to iteratively improve Python and C++ programs Gao et al. (2025). POLO addresses project-level optimization by profiling hotspots,

constructing program-structure graphs, and coordinating multiple agents to rewrite related functions Bai et al. (2025). LLMs have also been applied to individual optimization domains, including verified loop vectorization Taneja et al. (2025), LLM-assisted source transformations for auto-vectorization Zheng et al. (2025), and code generation for tensor accelerators Hong et al. (2025). Because generated transformations may violate program semantics, recent systems incorporate formal or runtime verification. LLM-VeriOpt uses Alive2 feedback during reinforcement learning Fang et al. (2026), whereas CoV combines static, symbolic, and runtime checks Kwon et al. (2026). AICORE applies the same reference-output validation process to every candidate. Its pass rate measures agreement on the evaluated executions rather than formal equivalence.

Another line uses LLMs to guide compiler decisions. Reasoning Compiler uses an LLM to explore context-dependent optimization decisions for model serving Tang et al. (2025), while Compiler-R1 trains an agent to interact with compiler tools and generate LLVM pass sequences Pan et al. (2025a). ECCO combines evidence-based causal reasoning with evolutionary search for compiler auto-tuning Pan et al. (2026). Garber et al. (2025) use an LLM to select GCC optimization options for all 30 PolyBench kernels. AutoPass exposes compiler-internal states and IR to an LLM that iteratively selects compiler optimization configurations Li et al. (2026). AICORE also uses compiler evidence, but its action space includes source rewrites and loop pragmas alongside parameters. Because no official AutoPass implementation is publicly available, we reimplemented it using the parameters reported in its paper; we did not run the other systems in this paragraph.

Learning-based compiler auto-tuning.

Before language models, compiler tuning was formulated as search over optimization configurations, with methods differing in their search strategy, program representation, and organization of the search space Ashouri et al. (2019). Iterative compilation directly evaluates sampled flags or pass sequences Kulkarni et al. (2004; 2007), while evolutionary Hoste & Eeckhout (2008), polyhedral Pouchet et al. (2008), and Monte Carlo tree search Koo et al. (2021) provide alternative exploration strategies. Surrogate-based methods reduce costly measurements: BOCA uses a random-forest Bayesian model and measured flag importance Chen et al. (2021), whereas Bliss maintains a diverse pool of lightweight Bayesian models Roy et al. (2021).

Program-characterization methods predict or guide optimization choices. Early models learned individual heuristics Stephenson & Amarasinghe (2005); Cavazos & O'Boyle (2006), followed by approaches using static features and performance counters Agakov et al. (2006); Cavazos et al. (2007); Fursin et al. (2011). Later systems transfer knowledge across programs: Reaction Matching recommends configurations from responses to optimization sets Cereda et al. (2020); ICMC combines metric learning and collaborative filtering to prune features and derive pass subsequences Liu et al. (2021); and long-term learning evolves heuristics across successive compilations without a separate training phase Tartara & Crespi Reghizzi (2013).

Other systems restructure the search space around pass interactions. MiCOMP clusters `-O3` passes into subsequences Ashouri et al. (2017); SRTuner models influential pass relations as bandit problems Park et al. (2022); and Pan et al. (2025b) form coresets from synergistic pass pairs. AutoPhase learns sequential pass selection through deep reinforcement learning Haj-Ali et al. (2020), whereas Liang et al. (2023) select complete sequences from a coreset. MLGO learns compiler heuristics such as inlining for size Trofin et al. (2021), and ACPO integrates learned unrolling and inlining decisions into LLVM Ashouri et al. (2023). Input sensitivity complicates evaluation Chen et al. (2010), while compilation statistics can guide tuning Zhao et al. (2025). Unlike these compiler-focused methods, AICORE evaluates source rewrites and compiler controls within the same validated candidate loop.

Compiler–LLM cooperation and verification. Mikek et al. (2026) coordinate source, IR, and assembly agents with compiler tools and test generation. T-LLM Compiler combines source optimization with syntax, symbolic, and semantic verification and evaluates the same 30 PolyBench/C kernels on ARM Fazel et al. (2026). ComPilot uses compiler legality and performance feedback to guide PolyBench loop schedules Merouani et al. (2025); HintPilot synthesizes compiler hints and evaluates them on PolyBench and HumanEval-CPP Jiang et al. (2026). AICORE refreshes compiler evidence for the retained source and combines source, pragma, and parameter actions, but these systems have not been compared under the same environment. PerfCodeBench offers executable systems-optimization tasks Jing et al. (2026); selected compatible tasks could test generality beyond the present suites.

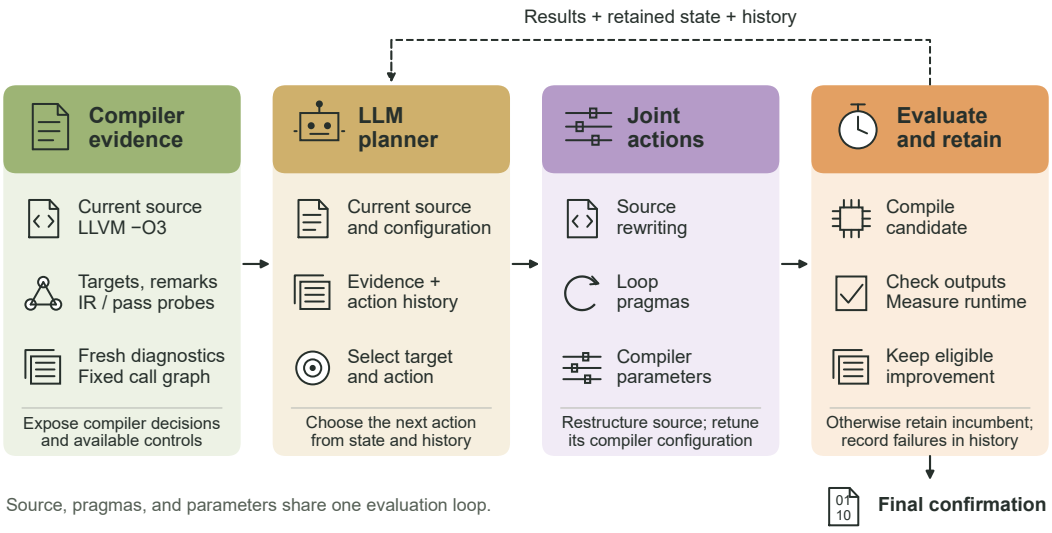


Figure 1: **AICORE overall workflow.**

3 METHOD

3.1 MOTIVATION EXAMPLE

PolyBench's BiCG kernel illustrates how compiler diagnostics can guide an LLM toward a useful source transformation. Clang reports that the inner loop was not vectorized and that loop-invariant loads could not be hoisted because the loop might invalidate their values. The LLM responds by loading `r[i]` into a local scalar and accumulating `q[i]` in another scalar, reducing the aliasing uncertainty seen by the compiler; it then adds a loop-local vectorization pragma. Recompilation confirms that the transformation is applied, and the retained candidate reaches $1.67\times$ speedup over `-O3`. This example shows how compiler feedback can narrow the source-level search, while validation and measurement determine whether the proposed change should be retained. Appendix Figure 4 provides a detailed view of the compiler feedback and a related compiler-guided BiCG candidate.

3.2 OVERALL ARCHITECTURE

Figure 1 summarizes the iterative workflow of AICORE. In each round, AICORE recompiles the retained program and collects up-to-date, source-linked compiler feedback: optimization remarks, relevant LLVM IR excerpts, records of applied and missed transformations, and supported optimization parameters (Section 3.3). The LLM planner combines this compiler feedback with the current candidate and action history to choose a source rewrite, loop pragma, or compiler-parameter change (Section 3.4). Every candidate then enters the same compile–validate–measure loop: eligible improvements replace the incumbent, while rejected attempts leave it unchanged but remain in the history. The resulting compiler and runtime feedback guides the next round, allowing the generated code to be refined against the behavior of the current compiled program (Section 3.5).

3.3 AI NATIVE COMPILER

In AICORE, the *AI-native compiler* is an agent-facing layer built on existing compiler facilities. It provides three functions. First, a project analyzer identifies functions and code regions that are likely to dominate execution time. Second, a query interface accepts a source region and returns structured, source-linked reports describing which transformations the compiler applied, missed, or could not legally perform. These reports help the LLM distinguish opportunities that require a source rewrite from those that can be addressed through compiler guidance. Third, an action interface allows the agent to apply localized pragmas and supported compiler parameters that influence subsequent transformations.

Project-level hot-region analyzer. Starting from `main`, AICORE traverses the reachable call graph and ranks functions using structural signals such as loop participation, arithmetic density, and I/O behavior. Compiler-generated runtime profiles identify which functions are active for the evaluated workload, allowing the analyzer to discard inactive code and prioritize high-ranked regions. This process reduces a large project to a compact set of optimization targets for the LLM planner.

Region-specific white-box compiler reports. The query interface accepts a file name and source-line range from the agent, then aggregates compiler analysis for that region across multiple optimization passes. It summarizes relevant data-flow and control-flow properties, identifies transformations that were successfully applied, and—most importantly—explains which transformations were blocked or rejected and why.

Raw compiler output is poorly suited to this interaction because a pass may emit a separate remark each time it analyzes the code, producing large and repetitive reports. For example, the 36-line `2mm` PolyBench kernel generates more than 1,300 lines of pass remarks, totaling roughly 10,000 tokens. This volume can obscure the small number of observations that are useful for deciding how to rewrite the program.

To reduce this noise, the interface organizes each optimization into three outcomes. First, optimizations that are irrelevant to the selected region are omitted; for example, loop-vectorization information is not reported for a region without a loop. Second, successfully applied optimizations are summarized with their remarks and relevant IR excerpts, helping the LLM avoid duplicating work already performed by the compiler. Third, failed or blocked optimizations are reported together with the compiler's explanation, allowing the LLM to combine this information with the source code and determine whether a rewrite, pragma, or parameter change could enable the transformation.

3.4 Joint source rewrites and compiler actions

The AI-native compiler first narrows the program to active optimization targets and produces a region-specific report for each target. The planner combines these reports with the retained source, current compiler configuration, and outcomes of previous attempts to select one of three complementary action channels. All three channels modify the same retained candidate, allowing later actions to build on earlier improvements.

(1) Compiler pragmas. Pragmas request a transformation at a specific source location, providing fine-grained guidance without affecting unrelated code. For example, `#pragma clang loop unroll_count(4)` asks Clang to unroll the selected loop by a factor of four. **(2) Compiler parameters.** Parameters adjust broader decisions such as cost models and optimization effort. AICORE discovers settings supported by the installed compiler, screens individual parameters, and then evaluates promising combinations. **(3) Source rewrites.** When a transformation is blocked by the program structure, the planner generates a localized edit, such as loop tiling or interchange to improve data locality. The program is then recompiled to determine whether the rewrite enabled the intended compiler transformation.

Regardless of the action channel, the resulting candidate enters the shared compile–validate–measure loop described next. An accepted candidate becomes the retained state for the following round; otherwise, the previous state is preserved and the failed attempt remains in the planner's history.

3.5 Search, validation, and candidate retention

Algorithm 1 implements the feedback loop in Figure 1. It maintains the retained candidate z^*, the call graph G, and an action history H. At each round, AICORE collects fresh compiler feedback from z^*, selects an action using the current state and history, and evaluates the resulting candidate in an isolated workspace. Early rounds cover the three action channels, while later rounds use previous successes and failures to prioritize promising actions and avoid repetition.

Unified compiler-guided search. Algorithm 1 searches source rewrites, localized compiler pragmas, and compiler command-line options within a single candidate space, allowing improvements from different channels to be composed. At the start of each round, AICORE recompiles the retained candidate and uses the resulting compiler report to reduce this otherwise large search space:

Algorithm 1 AICORE: joint optimization with checked candidate retention

Require: Original program z_0, validation inputs $\mathcal{D}_{\text{check}}$, round budget R
Record reference output and validation status
$z^* \leftarrow z_0$; $H \leftarrow \varnothing$; $G \leftarrow \textsc{CallGraph}(z_0)$
for $t = 1, \ldots, R$ **do**
  $C_t \leftarrow \textsc{CollectCompilerFeedback}(z^*, G)$
  $a \leftarrow \textsc{ScheduleOrPlan}(t, z^*, C_t, H)$
  if $a = \text{stop}$ **then**
    break
  end if
  $q \leftarrow \textsc{Snapshot}(z^*)$
  $(z', o) \leftarrow \textsc{ExecuteAction}(a, z^*, C_t)$
  $H \leftarrow \textsc{Append}(H, (a, o))$ ▷ Include failures and regressions
  if z' builds, matches the reference output, and improves the screening score **then**
    $z^* \leftarrow z'$
  else if the action causes a severe regression **then**
    Restore working files from q; retain H
  else
    Discard z'; retain z^* and H
  end if
end for
Evaluate the final source–pragma–parameter composition with the same rule
return z^* and $\textsc{FinalRemeasurement}(z_0, z^*)$

unsupported or irrelevant controls are removed, transformations already applied by the compiler are deprioritized, and failed or blocked transformations identify promising rewrites, pragmas, or parameter changes. The planner then selects an action using the current candidate and history H, applies it in an isolated snapshot, compiles the result, compares its output with the original executable, and measures its screening performance. A candidate replaces the incumbent only if it builds, passes validation, and improves the screening score; otherwise, it is discarded while its outcome remains in H to prevent repeated failures. Each accepted candidate is recompiled to refresh the compiler report for the next round. After the search terminates, the final source–pragma–parameter composition is independently remeasured using the protocol in Section 4; this final measurement is reported in the evaluation. Output matching establishes agreement only for the evaluated executions rather than formal equivalence.

4 Experiments

4.1 Experimental setup

We evaluate all 30 numerical kernels from PolyBench/C 4.2 (Pouchet & Yuki, 2016) and 29 programs from cBench (cTuning community, n.d.). Experiments use two identically configured NVIDIA DGX Spark systems, each equipped with a 20-core Arm CPU (Cortex-X925/A725) and 119 GiB of RAM, running Ubuntu 24.04.4 and LLVM 21.1.8. AICORE and OpenCode both use DeepSeek-V4-Pro (DeepSeek-AI et al., 2026) with temperature 0.60, holding the LLM backend and sampling temperature constant in their comparison. Runs execute serially on each machine without explicit frequency locking. Time is measured relative to the original `-O3` executable, while validation independently compares candidate and reference outputs. Appendix A details the setup. Each measurement discards one warm-up run and reports the median of five trimmed means, each computed from four executions after removing the fastest and slowest.

Baselines and ablations. We use the open-source general-purpose coding agent OpenCode (Anomaly Innovations, 2025) as a baseline. AutoPass (Li et al., 2026) represents evidence-guided compiler tuning; because no official implementation is publicly available, we use our reimplementation configured with the parameters reported in the paper. We additionally report instrumentation-based PGO (Instr.PGO) (LLVM Project, n.d.) and OpenTuner (Ansel et al., 2014); the latter uses a 300-second search budget. We evaluate *rewrite-only*, *parameter-only*, and *without*

Table 1: **Per-program external comparison.** Values are speedups over `-O3`; AM is their arithmetic mean and Pass is the validation pass rate. X marks an invalid output. For aggregate AM computation, its retained `-O3` fallback contributes $1.00\times$; invalid outputs are excluded when determining the highest and second-highest valid result for each program. Bold marks the highest valid value and underlining the second highest.

PolyBench (1/2)

Program	AutoPass (reimpl.)	OpenCode	AICORE
2mm	1.01	**15.22**	9.82
3mm	1.00	**16.61**	6.85
adi	1.00	1.83	**3.23**
atax	1.00	0.46	**1.07**
bicg	1.00	1.47	**1.67**
cholesky	1.02	**1.85**	1.05
correlation	X	X	**13.61**
covariance	0.99	**29.51**	16.37
deriche	1.00	**3.10**	2.08
doitgen	1.02	**8.03**	4.90
durbin	1.02	**1.66**	1.32
fdtd-2d	1.03	X	**1.13**
floyd-warshall	X	**6.36**	5.69
gemm	1.00	**3.26**	1.47
gemver	1.00	0.66	**1.55**

PolyBench (2/2)

Program	AutoPass (reimpl.)	OpenCode	AICORE
gesummv	1.00	1.13	**1.67**
gramschmidt	1.00	**19.17**	14.63
heat-3d	1.00	X	**1.55**
jacobi-1d	1.00	**1.13**	1.11
jacobi-2d	1.17	**2.07**	1.18
lu	1.03	**1.27**	1.19
ludcmp	1.01	X	**1.13**
mvt	0.99	X	**1.66**
nussinov	X	**6.55**	1.18
seidel-2d	1.00	**6.08**	1.01
symm	1.17	**29.77**	10.24
syr2k	1.00	X	**4.23**
syrk	1.20	**5.75**	3.87
trisolv	1.00	X	**1.06**
trmm	1.00	**19.90**	17.13
AM	1.02	**6.33**	4.49
Pass (%)	90.00	76.70	**100.00**

cBench (1/2)

Program	AutoPass (reimpl.)	OpenCode	AICORE
automotive_bitcount	1.49	**30.73**	6.75
automotive_qsort1	0.99	**3.43**	1.00
automotive_susan_corners	1.00	1.01	**1.07**
automotive_susan_edges	0.98	X	**1.10**
automotive_susan_smoothing	1.00	X	**1.17**
bzip2_decode	1.00	1.00	**1.16**
bzip2_encode	1.00	1.00	**1.31**
consumer_jpeg_c	1.00	**1.08**	1.01
consumer_jpeg_d	1.00	**1.22**	1.00
consumer_lame	1.00	**1.95**	1.03
consumer_mad	1.00	**1.08**	1.06
consumer_tiff2bw	1.00	1.00	**1.01**
consumer_tiff2dither	0.99	**1.06**	1.04
consumer_tiff2median	1.00	1.00	**1.02**
consumer_tiff2rgba	0.96	1.00	**1.03**

cBench (2/2)

Program	AutoPass (reimpl.)	OpenCode	AICORE
network_dijkstra	1.00	**1.00**	1.00
network_patricia	1.00	1.03	**1.03**
office_rsynth	1.00	**1.05**	1.00
office_stringsearch1	1.00	**1.25**	1.00
office_stringsearch2	1.00	1.00	**1.03**
security_blowfish_decode	1.00	**1.02**	1.01
security_blowfish_encode	1.00	X	**1.27**
security_rijndael_decode	1.01	X	**1.03**
security_rijndael_encode	1.00	0.97	**1.25**
security_sha	1.00	0.99	**1.24**
telecom_adpcm_c	1.00	1.11	**1.37**
telecom_adpcm_d	1.00	**1.01**	1.00
telecom_crc32	1.00	1.01	**1.49**
telecom_gsm	1.00	**1.93**	1.19
AM	1.01	**2.20**	1.30
Pass (%)	**100.00**	86.20	**100.00**

compiler evidence variants of AICORE. The last variant retains execution checks and runtime feedback while withholding compiler evidence. AICORE and OpenCode use nine decision or conversational rounds. Because action costs differ, these comparisons are not matched-compute experiments.

Metrics. For each program, speedup is the separately measured `-O3` runtime divided by the candidate runtime; we report the arithmetic mean across each suite and the percentage of candidates that match the reference output. Invalid candidates are marked X and contribute $1.00\times$ through the retained `-O3` fallback, so the mean reflects safe end-to-end performance; all 59 AICORE candidates passed validation and improved over `-O3`.

4.2 Overall results

Validation pass rate. AICORE produced a candidate that matched the reference output for every evaluated program (Table 1). OpenCode failed validation on 23.33% of PolyBench kernels and 13.79% of cBench programs. Our AutoPass reimplementation achieved pass rates of 90.00% and 100.00%, respectively. These rates measure output agreement on the evaluated executions.

Performance speedup. AICORE reached $4.49\times$ speedup on PolyBench and $1.30\times$ on cBench, versus $1.02\times$ and $1.01\times$ for our AutoPass reimplementation. AICORE's joint action space exposes source transformations in addition to compiler controls, enabling optimizations unavailable to compiler-only baselines. OpenCode achieves higher arithmetic means ($6.33\times$, $2.20\times$), but lower validation pass rates. Its means are driven by several large successful rewrites—for example, $29.77\times$

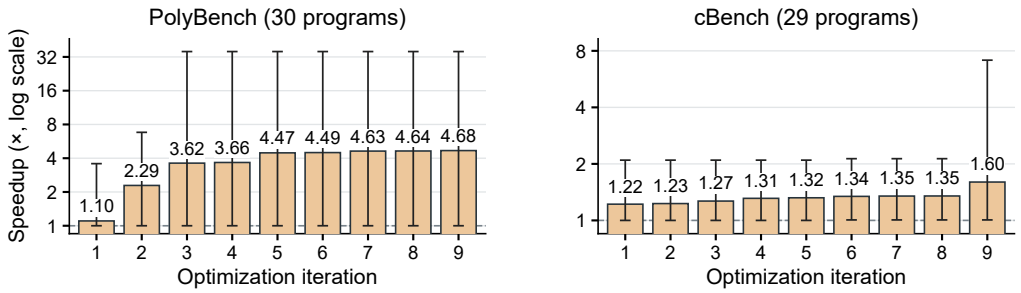


Figure 2: **Search progress over nine decision rounds.** Bars show the arithmetic mean of the best speedup retained by each round; whiskers span the minimum to maximum across programs on logarithmic axes. Missing entries retain the preceding value.

on `symm`, $29.51\times$ on `covariance`, and $30.73\times$ on `automotive_bitcount`—while seven PolyBench and four cBench outputs failed the available checks. AICORE trades a lower mean for a candidate passing the available checks on every program in the recorded evaluation. AICORE exceeds Instr.PGO ($1.04\times$, $1.00\times$) and OpenTuner ($1.09\times$, $1.14\times$) on the two suites.

For context, Garber et al. (2025) report a $1.02\times$ average for LLM-selected GCC options on all 30 PolyBench kernels, and T-LLM Compiler reports $1.18\times$ under its best-of-prompts setting (Fazel et al., 2026). These are results from the original papers rather than local baselines: their hardware, compilers, inputs, search procedures, and validation protocols differ from ours.

4.3 PolyBench ablation study

We use PolyBench's 30 loop-intensive kernels for component ablations and cBench only for heterogeneous end-to-end evaluation, so component-level conclusions are limited to PolyBench.

Ablation results. On PolyBench, AICORE achieves a $4.49\times$ arithmetic-mean speedup, compared with $3.50\times$ without compiler evidence, $3.01\times$ for rewrite-only, and $1.03\times$ for parameter-only. The corresponding validation pass rates are 100.00%, 96.67%, 100.00%, and 73.33%. These results show that source-enabled configurations outperform the tested parameter space alone and that the full system exceeds the no-compiler-evidence variant. Because the variants may generate different proposals, the comparison does not isolate the causal effect of compiler feedback, and passing the available checks does not establish semantic equivalence. Appendix A.3 reports all eight documented rewrite-only remeasurements.

4.4 Search progress

Improvements accumulate across decision rounds. Across the archived runs (Figure 2), the mean best-so-far speedup rises from $1.10\times$ to $4.68\times$ on PolyBench and from $1.22\times$ to $1.60\times$ on cBench. The traces show that later rounds continue to find useful candidates. The screening score is used only for candidate selection and can be optimistic under runtime noise; all reported results therefore use independent final remeasurement as shown in Table 1.

4.5 Case studies

Figure 3 summarizes the BiCG motivation example and a second PolyBench case, followed by a cBench case showing why source-level actions complement compiler parameters.

Expose a contiguous update loop. In `doitgen`, missed hoisting and vectorization remarks prompted exchanging p and s. The retained rewrite replaced strided `C4[s][p]` traversal with contiguous access and reused `A[r][q][s]` through a scalar. With a subsequent vectorization pragma, the final remeasurement reached $4.90\times$.

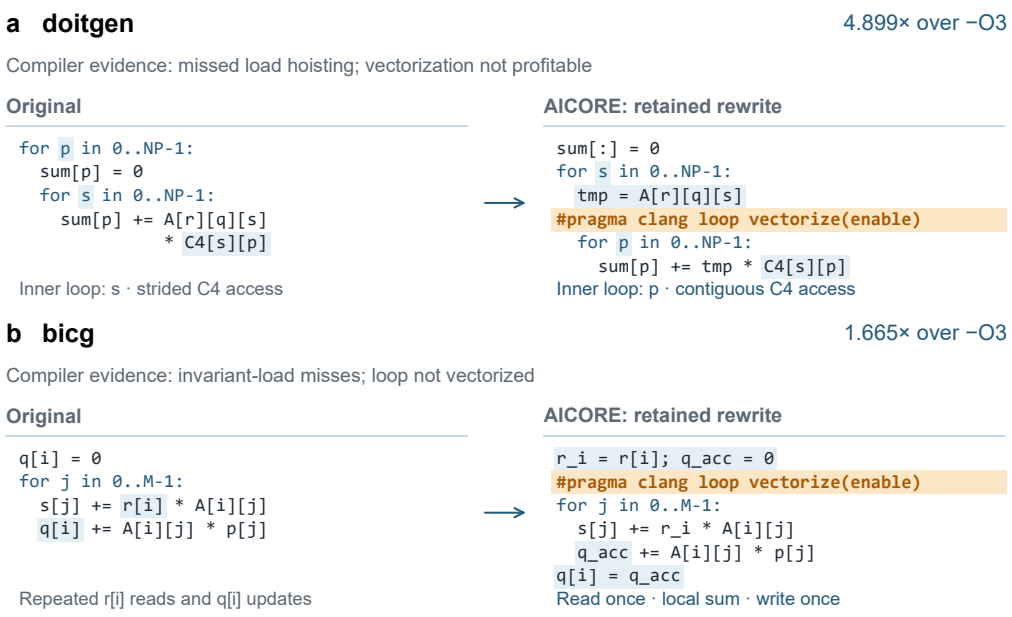


Figure 3: **Diagnostics motivate source changes.** Blue marks loop indices and scalar reuse; amber marks LLM-inserted pragmas. Scores are final remeasurements over `-O3`.

Remove application-level redundant work. The cBench program `automotive_bitcount` computes the same population count seven ways and accumulates identical results. AICORE calls `bitcount(seed)` once per input and reuses the result in the original update order, eliminating redundancy unavailable to pass-order or parameter tuning and reaching $6.75\times$.

5 Discussion and Limitations

The results support the central design choice: compiler evidence is most useful when it can trigger both compiler controls and source changes. Figure 3 shows how missed transformations motivate localized rewrites, while the cBench case exposes redundant work that compiler-only tuning does not remove. AICORE therefore exceeds the tested compiler-only configurations in arithmetic mean and the tested PolyBench variants in the recorded evaluation. The comparison does not isolate every component because the methods follow different search paths and incur different costs. Output validation covers the evaluated executions rather than all possible inputs, and benchmark programs do not establish project-level generality. These limits bound the claims without changing the observed speedups and validation rates.

6 Conclusion

AICORE connects compiler evidence to source, pragma, and parameter actions in a shared validated search loop. It reaches $4.49\times$ and $1.30\times$ arithmetic-mean speedup on PolyBench and cBench, and every reported candidate matches its reference output. Under the evaluated configurations, AICORE exceeds the compiler-only baselines in mean speedup, achieves a higher validation rate than OpenCode, and outperforms the tested PolyBench ablations.

Reproducibility Statement

Sections 3 and 4 describe the algorithm and evaluation, and Appendix A provides additional implementation, validation, timing, and environment details. An anonymized code snapshot and the per-program result tables are available at `https://anonymous.4open.science/r/283ee234ghq`.

AI USE STATEMENT

Generative AI is a component of the evaluated system: DeepSeek-V4-Pro serves as AICORE's inference-time planner (Section 4.1). Separately from this, in preparing this work we used generative AI tools to [implement parts of the AICORE framework and experiment scripts / reformat exported experimental records into tables]. We have not used generative AI tools to design the methodology or experiments, formulate claims, generate data, or interpret results. Additionally, we used generative AI tools for drafting and revising text, language polishing, preparing figures and tables, and checking references. All AI-assisted code was [reviewed and tested by the authors]; all reported numbers were produced by compiling, validating, and timing the programs and were cross-checked by the authors against the raw logs. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O'Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using machine learning to focus iterative optimization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pp. 295–305. IEEE Computer Society, 2006. doi: 10.1109/CGO.2006.37. URL https://doi.org/10.1109/CGO.2006.37.

Anomaly Innovations. OpenCode: The open source AI coding agent. https://github.com/anomalyco/opencode, 2025. MIT License.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pp. 303–316. ACM, 2014. doi: 10.1145/2628071.2628092. URL https://doi.org/10.1145/2628071.2628092.

Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. MiCOMP: Mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3), September 2017. ISSN 1544-3566. doi: 10.1145/3124452. URL https://doi.org/10.1145/3124452.

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, 2019. doi: 10.1145/3197978. URL https://doi.org/10.1145/3197978.

Amir H. Ashouri, Muhammad Asif Manzoor, Duc Minh Vu, Raymond Zhang, Colin Toft, Ziwen Wang, Angel Zhang, Bryan Chan, Tomasz S. Czajkowski, and Yaoqing Gao. ACPO: An AI-enabled compiler framework, 2023. URL https://arxiv.org/abs/2312.09982.

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*, pp. 7319–7328, 2025. doi: 10.24963/ijcai.2025/814. URL https://doi.org/10.24963/ijcai.2025/814.

John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In Peri L. Tarr and William R. Cook (eds.), *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pp. 229–240. ACM, 2006. doi: 10.1145/1167473.1167492. URL https://doi.org/10.1145/1167473.1167492.

John Cavazos, Grigori Fursin, Felix V. Agakov, Edwin V. Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*, pp. 185–197. IEEE Computer Society, 2007. doi: 10.1109/CGO.2007.32. URL https://doi.org/10.1109/CGO.2007.32.

Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '20, pp. 15–25, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370943. doi: 10.1145/3372799.3394361. URL https://doi.org/10.1145/3372799.3394361.

Hongzheng Chen, Alexander Novikov, Ngân Vũ, Hanna Alam, Zhiru Zhang, Aiden Grossman, Mircea Trofin, and Amir Yazdanbakhsh. Magellan: Autonomous discovery of novel compiler optimization heuristics with alphaevolve, 2026. URL https://arxiv.org/abs/2601.21096.

Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pp. 1198–1209. IEEE Press, 2021. ISBN 9781450390859. doi: 10.1109/ICSE43902.2021.00110. URL https://doi.org/10.1109/ICSE43902.2021.00110.

Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In Benjamin G. Zorn and Alex Aiken (eds.), *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pp. 448–459. ACM, 2010. doi: 10.1145/1806596.1806647. URL https://doi.org/10.1145/1806596.1806647.

cTuning community. Collective benchmark (cBench). cTuning project documentation, n.d. URL https://ctuning.org/wiki/index.php/CTools:CBench. Accessed September 15, 2026.

DeepSeek-AI, Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026. doi: 10.48550/arXiv.2606.19348. URL https://arxiv.org/abs/2606.19348.

Shouyang Dong, Jun Bi, Di Huang, Jiaming Guo, Jianxing Xu, Ruibai Xu, Xinkai Song, Yifan Hao, Ling Li, Xuehai Zhou, et al. {QiMeng-Xpiler}: Transcompiling tensor programs for deep learning systems with a {Neural-Symbolic} approach. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pp. 239–255, 2025.

Xiangxin Fang, Jiaqin Kang, Rodrigo Rocha, Sam Ainsworth, and Lev Mukhanov. LLM-veriopt: Verification-guided reinforcement learning for LLM-based compiler optimization. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 740–755. IEEE, 2026. doi: 10.1109/CGO68049.2026.11395239. URL https://doi.org/10.1109/CGO68049.2026.11395239.

Zahra Fazel, Sunanda Gamage, Shayan Shirahmad Gale Bagi, Amir H. Ashouri, Tomasz S. Czajkowski, Bryan Chan, Reza Azimi, and Yaoqing Gao. T-LLM compiler: Trusted LLM-based code optimization and verification framework, 2026. URL https://arxiv.org/abs/2608.14953.

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. Milepost GCC: machine learning enabled self-tuning compiler. *Int. J. Parallel Program.*, 39(3):296–327, 2011. doi: 10.1007/S10766-010-0161-2. URL https://doi.org/10.1007/s10766-010-0161-2.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. Search-based LLMs for code optimization. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 578–590. IEEE, 2025. doi: 10.1109/ICSE55347.2025.00021. URL https://doi.org/10.1109/ICSE55347.2025.00021.

Damian Garber, Tamim Burgstaller, Sebastian Lubos, Patrick Ratschiller, and Alexander Felfernig. Towards LLM-enhanced compiler optimization. In *Proceedings of the 27th International Workshop on Configuration (ConfWS 2025)*, volume 4149 of *CEUR Workshop Proceedings*, pp. 62–69, 2025. URL https://ceur-ws.org/Vol-4149/paper5.pdf.

Ameer Haj-Ali, Qijing (Jenny) Huang, William S. Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (eds.), *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/hash/5b47430e24a5a1f9fe21f0e8eb814131-Abstract.html.

Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. Autocomp: A powerful and portable code optimizer for tensor accelerators. *arXiv preprint arXiv:2505.18574*, 2025. URL https://arxiv.org/abs/2505.18574.

Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Mary Lou Soffa and Evelyn Duesterwald (eds.), *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, pp. 165–174. ACM, 2008. doi: 10.1145/1356058.1356080. URL https://doi.org/10.1145/1356058.1356080.

Hanyun Jiang, Peisen Yao, Kaiyue Li, Tingting Lin, Chengpeng Wang, and Kui Ren. HintPilot: LLM-based compiler hint synthesis for code optimization. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 24970–24986. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.1251. URL https://aclanthology.org/2026.findings-acl.1251/.

Huihao Jing, Wenbin Hu, Shaojin Chen, Haochen Shi, Hanyu Yang, Sirui Zhang, Haoran Li, and Yangqiu Song. PerfCodeBench: Benchmarking LLMs for system-level high-performance code optimization, 2026. URL https://arxiv.org/abs/2605.15222.

Jaehoon Koo, Prasanna Balaprakash, Michael Kruse, Xingfu Wu, Paul D. Hovland, and Mary W. Hall. Customized monte carlo tree search for LLVM/polly's composable loop optimization transformations. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2021), St. Louis, MO, USA, November 15, 2021*, pp. 82–93. IEEE, 2021. doi: 10.1109/PMBS54543.2021.00015. URL https://doi.org/10.1109/PMBS54543.2021.00015.

Prasad A. Kulkarni, Stephen Hines, Jason Hiser, David B. Whalley, Jack W. Davidson, and Douglas L. Jones. Fast searches for effective optimization phase sequences. In William W. Pugh and Craig Chambers (eds.), *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pp. 171–182. ACM, 2004. doi: 10.1145/996841.996863. URL https://doi.org/10.1145/996841.996863.

Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Evaluating heuristic optimization phase order search algorithms. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*, pp. 157–169. IEEE Computer Society, 2007. doi: 10.1109/CGO.2007.9. URL https://doi.org/10.1109/CGO.2007.9.

Hyunho Kwon, Sanggyu Shin, Ju Min Lee, Hoyun Youm, Seungbin Song, Seongho Kim, Hanwoong Jung, Seungwon Lee, and Hanjun Kim. Compiler-runtime co-operative chain of verification for LLM-based code optimization. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 617–629. IEEE, 2026. doi: 10.1109/CGO68049.2026.11395240. URL https://doi.org/10.1109/CGO68049.2026.11395240.

Zepeng Li, Jie Ren, Zhanyong Tang, Jie Zheng, and Zheng Wang. AutoPass: Evidence-guided LLM agents for compiler performance tuning, 2026. URL https://arxiv.org/abs/2606.20373.

Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh James Leather, and Yuandong Tian. Learning compiler pass orders using coreset and normalized value prediction. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 20746–20762. PMLR, 2023. URL https://proceedings.mlr.press/v202/liang23f.html.

Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. Iterative compilation optimization based on metric learning and collaborative filtering. *ACM Trans. Archit. Code Optim.*, 19(1), December 2021. ISSN 1544-3566. doi: 10.1145/3480250. URL https://doi.org/10.1145/3480250.

LLVM Project. Clang compiler user's manual: Profiling with instrumentation, n.d. URL https://clang.llvm.org/docs/UsersManual.html. Accessed September 18, 2026.

Massinissa Merouani, Islem Kara Bernou, and Riyadh Baghdadi. Agentic auto-scheduling: An experimental study of LLM-guided loop optimization. In *2025 34th International Conference on*

Parallel Architectures and Compilation Techniques (PACT), pp. 186–200. IEEE, 2025. doi: 10.1109/PACT65351.2025.00027. URL https://doi.org/10.1109/PACT65351.2025.00027.

Benjamin Mikek, Danylo Vashchilenko, Bryan Lu, and Panpan Xu. Agentic code optimization via compiler-LLM cooperation, 2026. URL https://arxiv.org/abs/2604.04238.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards agentic compiler auto-tuning with reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 88289–88311, 2025a. doi: 10.52202/085713-2951. URL https://papers.nips.cc/paper_files/paper/2025/hash/7fc0aac0d700873ac520af9b05e3f2b6-Abstract-Conference.html.

Haolin Pan, Yuanyu Wei, Mingjie Xing, Yanjun Wu, and Chen Zhao. Towards efficient compiler auto-tuning: Leveraging synergistic search spaces. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, pp. 614–627, New York, NY, USA, 2025b. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708961. URL https://doi.org/10.1145/3696443.3708961.

Haolin Pan, Lianghong Huang, Jinyuan Dong, Mingjie Xing, and Yanjun Wu. ECCO: Evidence-driven causal reasoning for compiler optimization. *arXiv preprint arXiv:2602.00087*, 2026. URL https://arxiv.org/abs/2602.00087.

Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. SRTuner: Effective compiler optimization customization by exposing synergistic relations. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, pp. 118–130. IEEE Press, 2022. ISBN 9781665405843. doi: 10.1109/CGO53902.2022.9741263. URL https://doi.org/10.1109/CGO53902.2022.9741263.

Louis-Noël Pouchet and Tomofumi Yuki. PolyBench/C 4.2: The polyhedral benchmark suite. Project website, 2016. URL https://sourceforge.net/projects/polybench/. Accessed September 23, 2026.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In Rajiv Gupta and Saman P. Amarasinghe (eds.), *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pp. 90–100. ACM, 2008. doi: 10.1145/1375581.1375594. URL https://doi.org/10.1145/1375581.1375594.

Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 1280–1295, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454109. URL https://doi.org/10.1145/3453483.3454109.

Mark Stephenson and Saman P. Amarasinghe. Predicting unroll factors using supervised classification. In *3rd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*, pp. 123–134. IEEE Computer Society, 2005. doi: 10.1109/CGO.2005.29. URL https://doi.org/10.1109/CGO.2005.29.

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*, pp. 137–149. ACM, 2025. doi: 10.1145/3696443.3708929. URL https://doi.org/10.1145/3696443.3708929.

Annabelle Sujun Tang, Christopher Priebe, Rohan Mahapatra, Lianhui Qin, and Hadi Esmaeilzadeh. Reasoning compiler: LLM-guided optimizations for efficient model serving. In *Advances in Neural Information Processing Systems*, volume 38, pp. 106904–106930, 2025. doi: 10.52202/085713-3565. URL https://proceedings.nips.cc/paper_files/paper/2025/hash/99e49b207cb4f5c3b4c3b7da0261e8af-Abstract-Conference.html.

Michele Tartara and Stefano Crespi Reghizzi. Continuous learning of compiler heuristics. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400705. URL https://doi.org/10.1145/2400682.2400705.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a machine learning guided compiler optimizations framework. *CoRR*, abs/2101.04808, 2021. URL https://arxiv.org/abs/2101.04808.

Jiayu Zhao, Chunwei Xia, and Zheng Wang. Leveraging compilation statistics for compiler phase ordering. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2025, Milano, Italy, June 3-7, 2025*, pp. 533–545. IEEE, 2025. doi: 10.1109/IPDPS64566.2025.00054. URL https://doi.org/10.1109/IPDPS64566.2025.00054.

Zhongchun Zheng, Kan Wu, Long Cheng, Lu Li, Rodrigo C. O. Rocha, Tianyi Liu, Wei Wei, Jianjiang Zeng, Xianwei Zhang, and Yaoqing Gao. VecTrans: Enhancing compiler auto-vectorization through LLM-assisted code transformations. *arXiv preprint arXiv:2503.19449*, 2025. URL https://arxiv.org/abs/2503.19449.

Qirui Zhou, Yuanbo Wen, Ruizhi Chen, Ke Gao, Weiqiang Xiong, Ling Li, Qi Guo, Yanjun Wu, and Yunji Chen. Qimeng-GEMM: Automatically generating high-performance matrix multiplication code by exploiting large language models. In Toby Walsh, Julie Shah, and Zico Kolter (eds.), *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, pp. 22982–22990. AAAI Press, 2025. doi: 10.1609/AAAI.V39I21.34461. URL https://doi.org/10.1609/aaai.v39i21.34461.

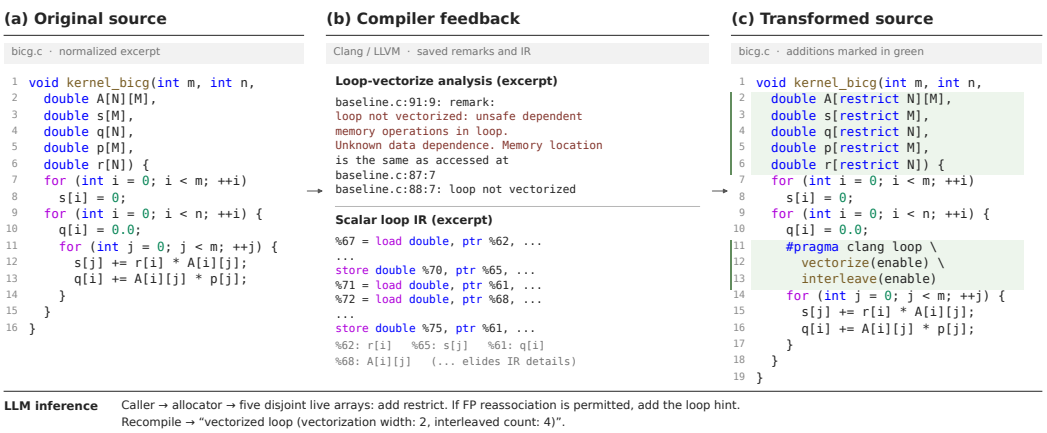


Figure 4: **Detailed BiCG compiler interaction.** The figure shows how dependence diagnostics motivate a restrict-based candidate and loop guidance. Figure 3 shows the separately retained scalar-hoisting rewrite used for the reported 1.67× result.

A Evaluation records and reproducibility

The evaluation contains 30 PolyBench and 29 cBench programs. The package includes every per-program result and its arithmetic-mean summary. Each reported AICORE candidate passed output comparison against the original execution. The search-progress figure reports archived search-time measurements; the case-study scores match the final per-program results in Table 1.

The main comparison applies the same validity rule wherever validation records are available. Invalid outputs are marked X, and the original `-O3` program is retained when computing the aggregate speedup. Appendix B reports the exported Instr.PGO and OpenTuner performance values separately because comparable validation counts were not retained for those baselines.

A.1 Validation, timing, and environment

Validation compares every candidate with the output of the original program. Discrete outputs use exact comparison, while numeric outputs use a tolerance-aware comparison. Failed executions or mismatched outputs are rejected. Validation and performance measurement use the same candidate source and compiler parameters.

The timing routine uses one warm-up followed by five groups of four executions. Within each group, the extreme values are discarded and the middle two averaged; the median across the five groups is returned. There are 21 executions per timing call under these defaults. These execution repetitions do not constitute independent optimization searches.

Both machines are bare metal with 20 cores, no simultaneous multithreading, 119 GiB memory, and Ubuntu 24.04.4 LTS. The target is `aarch64-unknown-linux-gnu`; Clang and opt are version 21.1.8. Reference builds use `-O3`; each candidate uses the same build and execution configuration except for its selected source changes and compiler controls.

The LLM endpoint used by AICORE and OpenCode is `https://api.deepseek.com`, with model `deepseek-v4-pro`, temperature 0.60, and output limit 4,000 tokens. Runs execute serially on each machine without explicit frequency locking. Token usage and exact compile counts were not retained, so the baseline comparisons should not be interpreted as matched-compute experiments.

A.2 Target ranking and planner context

Function ranking combines call-graph reachability, loop structure, arithmetic density, and an I/O penalty. Runtime activity checks prevent the planner from optimizing an inactive target. Together, these signals produce a compact set of active computation targets.

The planner context contains the current source, compiler remarks, IR probes, pass information, available counters, supported parameters, and candidate history. When the context limit is reached, current source and recent actionable evidence take priority. Candidate states, decisions, validation outcomes, and measurements are retained for audit.

A.3 DIRECTION-INDEPENDENT REMEASUREMENT ANALYSIS

The rewrite-only export contains eight documented five-run remeasurements. Replacing all eight program values with their recorded medians changes the full-cohort PolyBench arithmetic mean from $3.01\times$ to $3.08\times$, both below AICORE's $4.49\times$. This check replaces all eight records regardless of direction; it is not a matched-repeat comparison against AICORE.

Program	Exported value	Five-run median
correlation	9.66	9.66
durbin	1.52	2.16
fdtd-2d	1.02	1.28
floyd-warshall	1.76	1.76
gramschmidt	5.49	6.15
jacobi-1d	1.04	1.04
nussinov	1.07	1.29
syrk	2.51	2.84
Rewrite-only arithmetic mean (30)	3.01	3.08

Table 2: Rewrite-only PolyBench values and all eight documented remeasurement medians. The final row recomputes the arithmetic mean over all 30 programs.

B ADDITIONAL COMPILER BASELINES

Instr.PGO and OpenTuner each supply one exported result per program; OpenTuner uses a 300-second search limit. Their detailed training/search configurations and comparable validation counts are not available in the supplied table, so no pass percentage is inferred. Additional values have three-decimal source precision but are displayed to two decimal places; their arithmetic means are recomputed from the source values. AICORE retains the main-table results throughout.

B.1 POLYBENCH

Program	AICORE	Instr.PGO	OpenTuner
2mm	9.82	1.00	1.18
3mm	6.85	1.00	1.01
adi	3.23	1.00	1.01
atax	1.07	0.95	1.33
bicg	1.67	0.96	1.14
cholesky	1.05	0.94	1.00
correlation	13.61	1.00	1.01
covariance	16.37	1.00	1.00
deriche	2.08	0.90	1.16
doitgen	4.90	0.99	1.03
durbin	1.32	1.40	1.04
fdtd-2d	1.13	1.45	1.01
floyd-warshall	5.69	1.06	1.02
gemm	1.47	1.00	1.13
gemver	1.55	1.00	1.07
gesummv	1.67	1.06	1.20
gramschmidt	14.63	1.00	1.03
heat-3d	1.55	1.00	1.02
jacobi-1d	1.11	1.03	1.04
jacobi-2d	1.18	0.99	1.43
lu	1.19	1.00	1.00
ludcmp	1.13	1.00	1.06
mvt	1.66	0.97	1.05
nussinov	1.18	1.00	1.05
seidel-2d	1.01	1.00	1.03
symm	10.24	1.00	1.08
syr2k	4.23	0.99	1.00
syrk	3.87	0.99	1.11
trisolv	1.06	1.41	1.39
trmm	17.13	1.00	1.09
AM	4.49	1.04	1.09

B.2 cBENCH

Program	AICORE	Instr.PGO	OpenTuner
automotive_bitcount	6.75	1.39	1.06
automotive_qsort1	1.00	1.01	1.11
automotive_susan_corners	1.07	1.00	1.21
automotive_susan_edges	1.10	1.06	1.21
automotive_susan_smoothing	1.17	0.75	1.00
bzip2_decode	1.16	0.90	1.00
bzip2_encode	1.31	1.02	1.22
consumer_jpeg_c	1.01	0.77	1.06
consumer_jpeg_d	1.00	0.69	1.05
consumer_lame	1.03	1.00	1.05
consumer_mad	1.06	0.90	1.10
consumer_tiff2bw	1.01	1.07	1.07
consumer_tiff2dither	1.04	0.79	1.41
consumer_tiff2median	1.02	1.02	1.10
consumer_tiff2rgba	1.03	0.81	1.03
network_dijkstra	1.00	0.64	1.14
network_patricia	1.03	1.02	1.07
office_rsynth	1.00	0.57	1.06
office_stringsearch1	1.00	1.04	2.41
office_stringsearch2	1.03	1.01	1.12
security_blowfish_decode	1.01	1.00	1.04
security_blowfish_encode	1.27	1.64	1.03
security_rijndael_decode	1.03	2.46	1.08
security_rijndael_encode	1.25	0.80	1.14
security_sha	1.24	1.03	1.10
telecom_adpcm_c	1.37	1.01	1.14
telecom_adpcm_d	1.00	0.93	1.08
telecom_crc32	1.49	0.90	1.06
telecom_gsm	1.19	0.70	1.02
AM	1.30	1.00	1.14