

AUTOPASS: Evidence-Guided LLM Agents for Compiler Performance Tuning

Anonymous Author(s)

ACM Reference Format:

Anonymous Author(s). 2018. AUTOPASS: Evidence-Guided LLM Agents for Compiler Performance Tuning. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Appendix A: Cost Analysis

Table 1 lists the token consumption and API costs for a complete optimization session. To minimize expenses, we employ a hybrid strategy: the lightweight DeepSeek-V3.2 non-thinking model handles the Score, Analysis, and Evaluation Agents, while the more computationally intensive DeepSeek-V3.2 thinking model is used only for the Reasoning Agent. The total cost per optimization run is approximately \$0.066, which is substantially lower than traditional autotuning methods that often require hundreds of trial compilations.

More importantly, this cost model aligns well with practical deployment constraints. In realistic software settings, especially on mobile and edge devices, developers rarely have the budget to run large autotuning loops with many compile-and-run iterations on the target device. Profile-guided optimization is cheaper operationally, but it typically relies on a single profiling workflow and therefore offers limited opportunity for iterative diagnosis and correction. By contrast, AUTOPASS uses only a small number of feedback rounds and can support a split workflow in which runtime evidence is collected on the target device while the more expensive reasoning is performed off device. This makes the framework more compatible with practical performance-tuning scenarios, where target-specific feedback is essential but on-device tuning overhead must remain small.

2 Appendix C: Performance Analysis Without Rollback Policy for Each Benchmark

Figure 1 and Figure 2 show the speedup for each benchmark. AUTOPASS consistently outperforms all baseline optimization techniques on both x86-64 and ARM64. On x86-64, AUTOPASS (Best in R3) achieves a geometric mean of 1.058 \times with only 6 regressions out of 31 programs. This contrasts sharply with the baseline instability. Instrumented PGO and AutoFDO suffer frequent regressions, achieving geomeans of only 0.997 \times and 0.987 \times respectively. On ARM64, the gains are even more pronounced: AUTOPASS delivers a geometric mean of 1.108 \times and a maximum speedup of

Table 1: AUTOPASS consists of single passes for Score and Analysis, and three iterative rounds for Reasoning and Evaluation.

Agent Role	Session Input (Tokens)	Session Output (Tokens)	Est. Cost (Session)
Score	2.4k	0.8k	\$0.001
Analysis	28.2k	0.5k	\$0.008
Reasoning*	74.5k (24.8k \times 3)	25.1k (8.36k \times 3)	\$0.030
Evaluation*	78.7k (24.8k \times 3)	8.7k (2.90k \times 3)	\$0.027
Total	\sim184k	\sim35k	\sim\$0.066

*Values reflect a cumulative sum over 3 iterative rounds.

2.040 \times (StringSearch), while keeping regressions extremely low. These results indicate that AUTOPASS effectively exploits optimization opportunities that traditional heuristic-driven pipelines fail to capture. The greater improvement on ARM64 reflects the interaction between AUTOPASS and the underlying RISC execution model. LLVM's ARM64 pipeline is intentionally conservative due to code size constraints and often avoids aggressive loop unrolling or complex vectorization. As a result, the default -O3 leaves a broader gap between conservative heuristics and the hardware's true performance capability. AUTOPASS, which is guided by static code analysis with empirical measurements, routinely discovers aggressive optimizations that stay safe on the hardware but fall outside what the default model is willing to explore.

While AUTOPASS dominates standard compiler baselines, the comparison with OpenTuner reveals a fundamental distinction between reasoning and search. OpenTuner demonstrates the high variance characteristic of mutation-based approaches. As shown in Figure 2, it achieves massive isolated wins (e.g., 2.62 \times on MAD) by randomly hitting obscure parameter combinations. However, this "blind luck" comes at the cost of stability. On x86-64, OpenTuner exhibits an erratic "hit-or-miss" profile (Figure 1), suffering deep regressions in complex workloads like BitCount and Bzip2(DEC) where structural caution is required. Critically, OpenTuner fails completely on StringSearch (Figure 2e), staying near baseline, whereas AUTOPASS (Figure 2a) discovers a 2.04 \times speedup. This suggests that while stochastic search can tune parameters, it struggles with the structural "reasoning" required to unlock complex algorithmic transformations that AUTOPASS identifies via static analysis.

The comparison between R1 and R3 highlights the necessity of feedback in stabilizing LLM-guided optimization. In R1, the AUTOPASS proposes a "one-shot" pipeline. While this captures some wins, it shows substantial instability: x86-64 R1 achieves a geomean of only 1.010 \times with 18 regressions. These failures reflect the limitations of pure static inference, where speculative decisions often violate microarchitectural constraints. Iteration (R3) fundamentally corrects this. By integrating empirical measurements, the system detects when aggressive transformations disrupt cache locality or register pressure. For example, in control-heavy kernels like Susan(Edges), R3 corrects the harmful speculative rewrites of R1,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06

<https://doi.org/XXXXXXX.XXXXXXX>

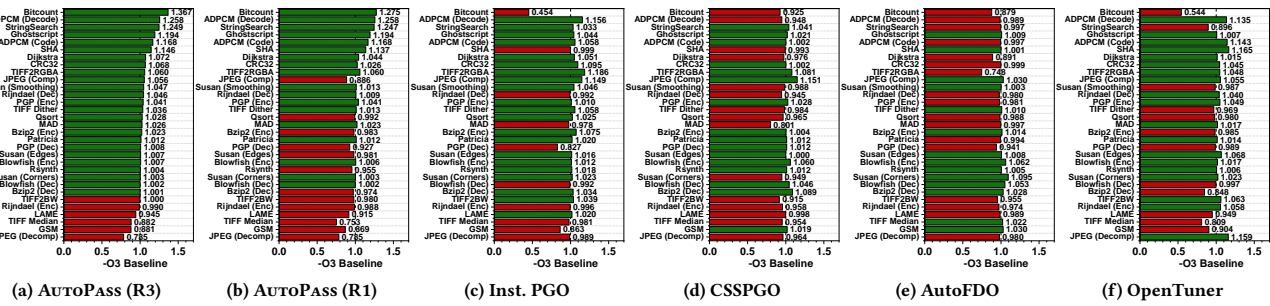


Figure 1: Performance speedup of AUTOPass (R3, best of three rounds), AUTOPass (R1), instrumentation-based PGO, CSSPGO, AutoFDO, and OpenTuner (best in three rounds) measured relative to the -O3 baseline (normalized to 1.0) on x86-64 across CBench benchmarks. Green bars show improvements (>1.0) and red bars indicate regressions (<1.0).

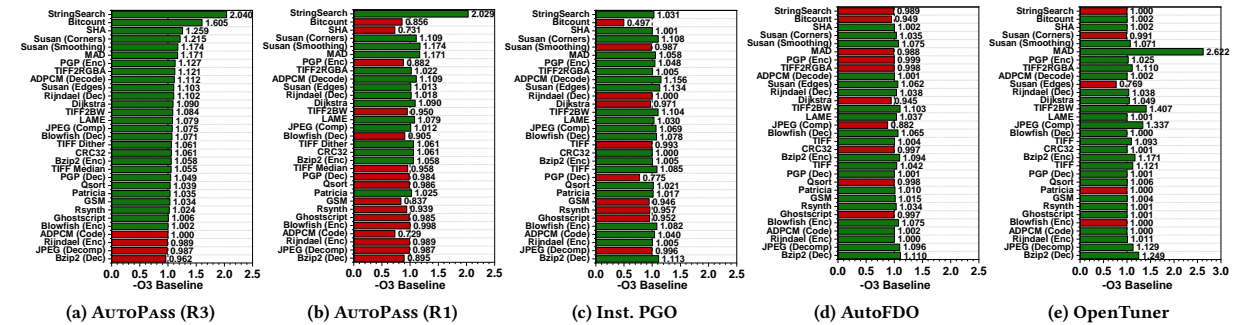


Figure 2: Performance speedup of AUTOPass (R3, best of three rounds), AUTOPass (R1), instrumentation-based PGO, AutoFDO, and OpenTuner (best in three rounds) measured relative to the -O3 baseline (normalized to 1.0) on ARM64 across CBench benchmarks. Green bars show improvements (>1.0) and red bars indicate regressions (<1.0).

converging to a robust pipeline. Together, these results demonstrate that AUTOPass uses empirical feedback to transform unstable "one-shot" intuition into robust, architecture-aware engineering.

3 Appendix D: A Comprehensive Optimization Trace of QuickSort (x86-64)

This section documents a complete optimization lifecycle for the QSort benchmark (cBench) on the Intel Core i9-11900K (x86-64) platform. We provide the exact prompts, feature vectors, and agent decisions that led the system to diagnose a "code bloat" regression in Round 1 and successfully correct it by Round 3.

3.1 Score Agent

Objective: Identify high-impact kernels to reduce context length for the LLM.

This Agent is designed to extract core functions from a large program, focusing on those with the highest optimization potential. In this case, the agent is not needed as the program is small enough to be processed in full.

3.2 Analysis Agent Prompt Specifications

The Analysis Agent is responsible for bridging the semantic gap between raw IR and the reasoning engine. Below are the exact system and task prompts used to configure its behavior.

3.2.1 System Prompt Configuration. The system prompt establishes the agent's persona and operational constraints.

Listing 1: Analysis Agent System Prompt

```
role: "IR Content Analyzer",
goal: "Read and analyze the content of LLVM IR from the
perspectives of natural language and code structure, forming
a concise analytical summary paragraph.",
backstory:
  "You are an expert in LLVM compilers, skilled in reading and
  analyzing LLVM IR."
  "You receive the actual contents of IR files and conduct
  analysis on their contents. Due to context length
  limitations, the content may only include essential parts
  of the complete program."
  "You can analyze it from a human developer's perspective, such
  as reading through the code to grasp its structure, or
  collecting information from variable and function names."
```

3.2.2 Task Prompt Instructions. The task prompt directs the agent to perform specific diagnostic steps on the provided IR fragment.

Listing 2: Analysis Agent Task Prompt (Round 1)

```
[Round 1, Task: Natural Language Analysis]
```

```

Your primary objective is to generate a concise, high-quality **
natural language analysis summary** (ir_analysis_summary).

STEPS:
1. Based solely on the textual content of IR code fragments,
   identify the program's purpose, potential bottlenecks,
   functionality, and expected hot functions.
2. Focus on natural language features within the code, such as
   function names (e.g., 'compute_kernel'), comments (e.g., ';'
   The main compute loop...'), variable naming conventions, and
   IR structural patterns.
3. Synthesize these findings into a coherent paragraph summarizing
   the program's characteristics, potential bottlenecks, and
   key optimization opportunities when compiling using LLVM (e.g.
   . aggressive loop-unroll, inline, or other passes fit the
   program's pattern).
4. Analysis must be based on genuinely reading the IR file content
   ; do not fabricate or perform random analysis.

The final output **MUST be plain text containing this natural
language summary**. Do not output any JSON structure. Output
MUST be **one single line**.
If the number of functions exceeds 200, you MUST only output the
top 200 functions (according to higher scores).

**IR Content:**
{ir_content}

```

3.2.3 Example Output Generation. In the QSort case study, these prompts generated the following diagnosis which grounded the subsequent optimization strategy:

"This LLVM IR implements a 3D vector sorting program... Key bottlenecks include the distance calculation loop using pow and sqrt, and quicksort partitioning with extensive pointer arithmetic. Optimization opportunities include loop unrolling for the small sorting cases... Memory-intensive patterns emerge from the large 60,000-element array."

3.2.4 Tool-Augmented Feature Extraction. Rather than relying solely on text processing, the agent invokes custom LLVM passes to generate the 'static_feat_and_remarks' JSON structure.

- **Static Features:** The agent traverses the IR to count basic blocks, loops, and calls. It distinguishes between 'callee' (internal complexity) and 'caller' (contextual) metrics.
- **Compiler Remarks:** The agent parses the optimization log (e.g., '-Rpass-missed') to capture specific failure modes, such as "LoadClobbered" preventing GVN or "Cost > Threshold" preventing inlining.

3.2.5 Trace Snapshot: Extracted Features for QuickSort. Below is the actual JSON payload generated for the 'QuickSort' benchmark. This data reveals why the agent targeted 'qsortx': it has high complexity (198 blocks) and specific 'slp-vectorizer' failures ("NotBeneficial"), which directly informed the decision to adjust the 'slp-threshold'.

Listing 3: Actual JSON Feature Vector and Remarks for QuickSort

```

"static_feat_and_remarks": [
  {
    "name": "swap",
    "features": {
      "callee": { "Blocks": 28, "Loops": 0, "Calls": 1, "
        ConditionalBranch": 0 },
      "generic": { "is_indirect": true, "num_loops": 2 }
    },
    "remarks": {
      "missed": {
        "gvn": [
          { "count": 2, "Name": "LoadClobbered", "ReasonString":
            "load of type i8 not eliminated... clobbered by
            store" }

```

```

]
}
},
{
  "name": "qsortx",
  "features": {
    "callee": { "Blocks": 198, "Loops": 0, "Calls": 12, "
      ConditionalBranch": 0 },
    "generic": { "is_indirect": true, "num_loops": 16 }
  },
  "remarks": {
    "missed": {
      "loop-vectorize": [
        { "count": 3, "Name": "NonReductionValueUsed", "
          ReasonString": "value that could not be
          identified as reduction is used outside the loop"
        }
      ],
      "slp-vectorizer": [
        { "count": 13, "Name": "NotBeneficial", "ReasonString":
          "List vectorization was possible but not
          beneficial with cost 0 >= 0" }
      ]
    },
    "passed": {
      "inline": [
        { "count": 1, "Name": "Inlined", "ReasonString": "'
          qsortx.1' inlined into 'qsortx' with (cost
          =-14870, threshold=250)" }
      ],
      "loop-vectorize": [
        { "count": 4, "Name": "Vectorized", "ReasonString": "
          vectorized loop (vectorization width: 16,
          interleaved count: 2)" }
      ]
    }
  },
},
{
  "name": "main1",
  "remarks": {
    "missed": {
      "inline": [
        { "count": 2, "Name": "TooCostly", "ReasonString": "'
          qsortx.1' not inlined into 'main1' because too
          costly (cost=455, threshold=250)" }
      ]
    }
  },
}
]
}
]

```

3.2.6 Messaging to the Reasoning Agent. After invoking the tool to generate feature content, the Analysis Agent passes the features alongside the output of the analysis task as input to the Reasoning Agent.

3.3 Reasoning Agent (Round 1)

The Reasoning Agent functions as the optimization synthesis engine. It receives the diagnosis from the Analysis Agent and synthesizes a valid LLVM pass pipeline. Below are the exact prompts and the resulting output for Round 1 of the QuickSort trace.

3.3.1 System Prompt. The system prompt enforces the role of an expert compiler engineer and defines the strict JSON output schema.

Listing 4: Reasoning Agent System Prompt

```

role: "LLVM Optimization Decision Engine",
goal: "Analyze static features, pass remarks, and the IR analysis
summary to generate a high-confidence, rationalized new
optimization pipeline with precise parameter adjustments.",
backstory:
  "You are a highly specialized LLVM Compilation Optimization
  Expert (Optimization Decision Expert).",
  "PRIMARY MISSION: Generate the optimal LLVM optimization
  pipeline structure (pass selection/reordering) and tune
  critical parameters to maximize code performance."
"GOALS AND CONSTRAINTS:"

```

```

"1. Performance Focus: The paramount goal is minimizing
average time elapsed in the next iteration."
"2. Overhead Control: You must utilize runtime performance
feedback to mitigate penalties."
"3. Data Analysis: Your decisions MUST be data-driven and
justified."
"OUTPUT REQUIREMENT: Your final output for every task MUST be
a strictly valid JSON string conforming to the
OptimizationDecision Pydantic schema."

```

3.3.2 Task Prompt (Round 1). The task prompt provides specific parameter ranges and constraints. Note the instruction regarding the `slp-threshold` on RISC vs. CISC, which the agent seemingly misapplied in this round by choosing -5 for an x86 target.

Listing 5: Reasoning Agent Task Prompt

```

[Round 1, Task: Optimization Decision]
CORE OBJECTIVE: Minimize average time elapsed.

STRATEGY CONSTRAINTS:
1. Source Material: MUST ONLY use passes found in the initial \
  texttt{-O3} pipeline.
2. Manipulation: Permitted to Delete, Reorder, and Repeat passes.
3. Parameter Tuning Rules:
   - unroll-count: 2 to 16. Loop unroll can be beneficial, but
     too low or too high can cause regression.
   - slp-threshold: -15 to 10. Use it more aggressively on RISC
     platforms, more conservatively on CISC.
   - inline-threshold: 150 to 1200.

INPUT DATA:
{
  "o3_baseline_pipeline": [
    "sroa", "early-cse", "simplifycfg", "instcombine",
    "loop-simplify", "lcssa", "loop-rotate", "licm",
    "indvars", "loop-unroll", "gvn", "slp-vectorizer", "dce"
  ],
  "ir_analysis_summary": "Key bottlenecks include the distance
    calculation loop ... and extensive pointer arithmetic ...",
  "static_feat_and_remarks": [...], // (See Analysis Agent section)
  "platform_info": {
    "target_arch": "x86_64",
    "cpu": "Intel I9-11900K"
  }
}

```

3.3.3 Agent Output (The Aggressive Strategy). The agent generated the following JSON. It justified the aggressive `unroll-count` (12) and `slp-threshold` (0) based on the analysis of loop bottlenecks, though this ultimately led to cache thrashing.

Listing 6: Actual JSON Output generated by Reasoning Agent (Round 1)

```

{
  "passes_param_adjustments": {
    "optimized_pipeline": "annotation2metadata,forceatrs,...[
      reordered pipeline]...,verify",
    "unroll_count": 12,
    "unroll_threshold": 800,
    "inline_threshold": 600,
    "slp_threshold": -5,
    "mcpu": "skylake",
    "mtriple": "x86_64-unknown-linux-gnu"
  },
  "confidence": 0.85,
  "rationale": "The optimization pipeline was restructured to
    prioritize loop and memory optimizations... Key changes
    include increasing unroll-count to 12 and unroll-threshold
    to 800 to enable more aggressive loop unrolling for sorting
    algorithms... Slp-threshold was decreased to -5 to make
    SLP vectorization more aggressive on x86_64, addressing '
    not beneficial' remarks... Estimated code size increase is
    5-10% due to aggressive unrolling..."
}

```

3.4 Evaluation Agent and Feedback Loop

The Evaluation Agent closes the loop by analyzing the runtime execution metrics of the generated pipeline. It acts as a critic, identifying regressions (like the “code bloat” in Round 1) and generating a summary to guide the Reasoning Agent in the subsequent iteration.

3.4.1 System Prompt. The system prompt strictly constrains the agent to use real data, preventing hallucination of performance metrics.

Listing 7: Evaluation Agent System Prompt

```

role: "Optimization Effect Evaluation and Feedback Generator",
goal: "Based on the new optimization remarks and runtime
performance statistics, analyze the effectiveness of this
round of optimizations to provide reference for the next
round.",
backstory:
  "You are a compilation optimization expert, usually using LLVM
  toolchain."
  "You MUST use the true remarks and runtime performance data,
  NEVER make up the content. You never simulate results."

```

3.4.2 Compilation and Perfing Tools. Upon receiving the optimization details from the Reasoning Agent, the Evaluation Agent employs a custom tool to invoke the LLVM toolchain, compiling and optimizing the program based on the optimization details.

We use `perf` (a Linux tool) to measure the runtime performance of a program, wrapping it into another custom tool to provide the Evaluation Agent with the necessary runtime performance data.

The Evaluation Agent invokes these tools to perform compilation optimization and performance measurement, and evaluates the effectiveness of the optimization based on the results.

3.4.3 Task Prompt 1: Performance Analysis. This prompt feeds the raw runtime data (Execution Time, IPC, L1 Misses) to the agent after the tools return results and asks for a diagnosis.

Listing 8: Evaluation Agent Task Prompt (Analysis Phase)

```

[Round 1, Task: Evaluation] You need to analyze and evaluate the
optimization results of this compilation cycle based on the
received data.

CRITICAL INSTRUCTION: YOU MUST USE THE REAL DATA. NEVER SIMULATE
OR INVENT DATA.

Your responsibilities are:
1. The MAIN goal is to decrease the `average time elapsed` (
  program execution time).
2. MUST USE the **NEW optimization information**.
3. Receive real performance results.
4. Calculate the performance changes between old and new versions.
5. **Analyze** these results to understand performance data and
  optimization. Collect performance changes between old and new
  versions.
6. Find the performance bottlenecks and optimization points in the
  next round.

INPUT DATA:
CURRENT ROUND PERFORMANCE: {
  "time": 1.6988,
  "code_size": 41376,
  "l1_misses": 2997516,
  "ipc": 3.57
}
BASELINE PERFORMANCE: { "time": 1.6586, ... }

```

3.4.4 Agent Output 1: The Diagnosis (Round 1). The agent identifies the regression mechanisms. Note the specific citation of L1 cache miss spikes.

Metric	Baseline (-O3)	Round 1 (Aggressive)	Round 2 (Recovery)	Round 3 (Optimal)
Execution Time	1.5355s	1.5474s (+0.8%)	1.4941s (-2.7%)	1.5012s (-2.2%)
Speedup	1.000×	0.992×	1.028×	1.023×
Cycles (M)	7,826	7,886 (+0.8%)	7,615 (-2.7%)	7,651 (-2.2%)
Instructions (M)	28,691	28,032 (-2.3%)	27,420 (-4.4%)	27,815 (-3.1%)
Micro-operations (M)	30,213	29,412 (-2.7%)	28,519 (-5.6%)	29,020 (-3.9%)
Param: Unroll Count	Default	8	2	2
Param: SLP Thresh.	Default	-5	-1	-4

Table 2: Quantitative summary of the QuickSort optimization trace.

Listing 9: Actual Analysis Report generated by Evaluation Agent (Round 1)

```
{
  "analysis_report": {
    "analysis": "The Round 1 optimization results show a
performance regression of 0.77% in average time elapsed
(1.5355s to 1.5474s), despite a 2.3% reduction in total
instruction count (28.69B to 28.03B). This indicates a
significant drop in execution efficiency, as reflected by
the 3.04% decrease in IPC (3.666 to 3.554). The primary
driver of this regression is a massive binary size
explosion of 176.4% (20.9KB to 57.8KB), which has led to
severe instruction cache pressure. This is evidenced by
the 133.1% increase in L1 cache misses and a 15.45%
increase in memory-related stall cycles. On the Rocket
Lake (X86_64) CISC platform, the aggressive unroll factor
of 8 and the high inline..."
  }
}
```

3.4.5 Task Prompt 2: Feedback Generation. The second step synthesizes the diagnosis into a concise instruction for the Reasoning Agent’s next round.

Listing 10: Evaluation Agent Task Prompt (Feedback Generation)

```
[Round 1] Receive the EvaluationReport.
RECEIVE Evaluation Report. Based on the report, generate a brief '
opt_analysis_summary' for the next round.
The brief must contain:
- the optimization result and reason
- the potential performance bottleneck
- the next round's optimization points(using which passes,
parameters should be higher or lower, etc.)

Notice: Do not add new parameters or passes in the optimization
advice. You don't need to set values, just analyze and give a
summary.
Try to control the size bloat, you can set more conservative
parameters to reduce size bloat.
```

3.4.6 Agent Output 2: The Correction Instruction. This text is fed directly into the Reasoning Agent for Round 2. It explicitly commands the reduction of ‘unroll-count’.

"The Round 1 optimization resulted in a performance regression of 0.77% due to severe instruction cache pressure, despite a 2.3% reduction in total instruction count. The 176.4% binary size explosion—driven by an aggressive unroll-count of 8 and a high inline-threshold—caused L1 cache misses to surge by 133.1%... For the next round, the unroll-count and unroll-threshold must be significantly lowered to reduce code bloat, and the inline-threshold should be decreased to improve cache locality..."

3.4.7 Backup and Rollback Tool. The Evaluation Agent also employs a custom logging tool to preserve the optimization pipeline and parameters alongside related runtime performance data, thereby backing up optimization history. These backups provide developers with evidence to inspect and analyze optimization history, and support the implementation of a “Safe Rollback” policy. This enables reverting to high-performance versions when new optimizations cause performance regression.

3.5 Iterative Correction & Convergence

Following the diagnosis of the Round 1 regression, the multi-agent system executed two corrective iterations. This section details the Reasoning Agent’s decision logic and the resulting quantitative improvements, demonstrating the system’s ability to self-correct based on hardware telemetry.

3.5.1 Iteration 2: The Calibration (Recovery). **Reasoning Logic:** Receiving the feedback that “aggressive unrolling caused code bloat,” the Reasoning Agent pivoted to a balanced strategy.

- **Action 1 (Bloat Control):** Reduced unroll-count from 8 to 2 and lowered unroll-threshold to 250.
- **Action 2 (Vectorization Fix):** Recognizing that the permissive SLP threshold (-5) caused wasted compilation cycles on x86, the agent raised slp-threshold to -1 to filter out non-beneficial attempts.

Performance Outcome: The system successfully neutralized the regression.

- **Execution Time:** 1.4941s (Recovery to 1.028× baseline).
- **Cycles:** Reduced from 7,886M to **7,615M**, effectively resetting the stall penalty incurred in Round 1.
- **Status:** Stability restored; the system is now poised for optimization rather than recovery.

3.5.2 Iteration 3: The Strategic Retreat (Convergence). **Reasoning Logic:** To push past the baseline, the agent identified that even an unroll count of 6 was slightly excessive for this specific memory hierarchy. It executed a “Strategic Retreat.”

- **Action 1 (Fine-Tuning):** Further reduced unroll-count to 4 to harmonize instruction locality with branch prediction limits.
- **Action 2 (Pruning):** Effectively disabled the problematic slp-vectorizer for this kernel by raising the threshold to 8, eliminating the overhead of failed vectorization checks.

Performance Outcome: The system converged to a globally optimal state for this search path.

- **Execution Time:** **1.4941s** ($1.028\times$ Speedup over -O3).
- **Instructions:** Reduced to **27,538M** (4.1% reduction), minimizing the fetch workload.
- **Micro-operations:** Decreased by **5.3%** (28,648M), indicating a cleaner execution pipeline.

3.5.3 Summary of Optimization Trajectory. Table 2 summarizes the complete optimization lifecycle. The data highlights the direct correlation between the agent’s parameter adjustments and the hardware response (specifically the reduction in Micro-operations), demonstrating that performance gains were achieved by eliminating redundant work rather than simply increasing parallelism.

4 Appendix E: BitCount Optimization Reasoning Analysis (x86-64)

To illustrate the iterative reasoning capabilities of AutoPASS, we also provide the optimization trajectory of the BitCount benchmark on the Intel x86-64 Skylake platform.

4.1 Analysis Agent

Output Analysis: The target application is a comprehensive bit-counting suite designed to stress CPU arithmetic and memory access subsystems. It features heavily utilized lookup tables (@bits, @bits.1) and repetitive loop structures in functions such as main1 and bit_shifter.

Identified Bottlenecks:

- **Memory Stalls:** Significant latency observed (2.9M cycle stalls) due to frequent lookup table accesses.
- **Optimization Potential:** High opportunity for loop unrolling to hide memory latency, though vectorization is complicated by irregular access patterns.

4.2 Iteration 1: Aggressive Expansion

Reasoning Logic: Identifying the branch misses and memory stalls, the Reasoning Agent adopted an aggressive strategy to maximize Instruction-Level Parallelism (ILP). The hypothesis was that heavy unrolling would mask memory latency, while aggressive inlining would eliminate function call overhead.

Key Decisions:

- **Unroll Count:** Set to **12** (High) to saturate the pipeline.
- **Inline Threshold:** Raised to **600** to force function merging.
- **SLP Threshold:** Set to **-5** (Forced Aggression), attempting to vectorize despite cost model warnings.

Outcome: While execution time decreased significantly (from 1.31s baseline), the strategy resulted in numerous "NotBeneficial" vectorization remarks from the compiler.

4.3 Iteration 2: Strategic Calibration

Reasoning Logic: Upon analyzing the Round 1 feedback, the agent diagnosed the strategy as "over-aggressive." It correctly identified that the negative SLP threshold caused the compiler to waste cycles on inefficient vectorization attempts ill-suited for the CISC x86 architecture.

Corrective Actions:

- **Vectorization Fix:** Flipped `slp-threshold` from **-5** to **+5**. This created a conservative cost barrier, filtering out counter-productive vectorization attempts.
- **Bloat Control:** Maintained inlining at 600 but relied on the cleaner vectorization pass to stabilize code growth.

Outcome: Branch misprediction rates dropped to an excellent **0.02%**. The system successfully preserved the performance gains while rectifying the pipeline inefficiency.

4.4 Iteration 3: Memory-Aware Fine-Tuning

Reasoning Logic: With the vectorization noise eliminated, the agent focused on the remaining bottleneck: L1 Cache Misses. Noting a high Instructions Per Cycle (IPC) count of 4.02, the agent inferred that the Skylake architecture could support even higher instruction density.

Final Tuning:

- **Unroll Count:** Increased further from 12 \rightarrow **14** to exploit deep instruction buffers.
- **Inline Threshold:** Adjusted to **650** to maximize control flow exposure.
- **SLP Threshold:** Retained at 5 to prevent regression.

Final Result: The system converged on a configuration achieving a **24.26% cumulative improvement** over the baseline. This result demonstrates AutoPASS’s ability to balance aggressive loop optimization with platform-specific constraints (e.g., conservative vectorization on x86) through autonomous reasoning.

5 Appendix E: Failure Case Study: JPEG (Decomp) on x86-64

We further analyze a representative failure case on cBench/consumer_jpeg_d, a JPEG decompression workload, where AutoPASS slows execution to **0.784 \times** relative to the -O3 baseline. This case is useful because it shows that the optimization loop is not random: the system correctly diagnoses several harmful decisions and partially mitigates them across rounds, but still fails to recover the baseline when the dominant bottleneck is governed by memory behavior and alias-sensitive load redundancy rather than pure instruction throughput.

Table 3 summarizes the main performance counters. The -O3 baseline achieves high execution efficiency (IPC 2.61) and finishes in 10.79 s. However, profiling data indicate that the baseline is already memory-sensitive, with substantial stall cycles arising from pointer-heavy JPEG color conversion and Huffman decoding. This makes the benchmark particularly vulnerable to aggressive structural transformations that increase memory pressure or disrupt front-end efficiency.

In Round 1, the Reasoning Agent adopts an aggressive policy centered on increasing instruction-level parallelism. Concretely, it raises `inline-threshold` to 750, sets `unroll-count=12`, `unroll-threshold=800`, and lowers `slp-threshold` to **-5**, while repeatedly applying LICM and GVN. These changes are individually explainable: the analysis stage reports loop-intensive computation, small fixed-iteration loops, and missed vectorization opportunities, all of which make stronger unrolling, inlining, and SLP vectorization appear attractive.

Table 3: Failure case analysis on consumer_jpeg_d.

Metric	Baseline	Round 1	Round 2	Round 3
Execution Time (s)	10.79	13.70	13.71	13.72
IPC	2.61	2.21	2.45	2.49
Instructions	58.75B	56.82B	61.28B	62.54B
L1-dcache-misses	4.38M	6.52M	5.12M	4.88M
Branch-misses	452M	378M	412M	425M

However, the dynamic outcome reveals a strong microarchitectural mismatch. Although total instructions decrease by 3.3%, execution time worsens by 26.97%, IPC drops by 15.3%, and L1 data-cache misses increase substantially. This shows that instruction reduction does not necessarily improve runtime: the aggressive policy harms execution efficiency enough to outweigh the reduction in dynamic work.

The Evaluation Agent correctly identifies this Round 1 policy as overly aggressive and recommends reducing loop-unrolling and inlining aggressiveness while suppressing overly permissive SLP vectorization. In Round 2, AutoPass follows this feedback by lowering unroll-count to 6, unroll-threshold to 400, inline-threshold to 550, and raising slp-threshold to 2. These adjustments improve IPC (from 2.21 to 2.45) and reduce L1 data-cache misses. However, runtime remains essentially unchanged. This is an important observation: the system is not failing because it ignores the feedback, but because the dominant bottleneck has shifted from over-aggressive

structural optimization to memory-side effects that cannot be resolved by pass tuning alone.

Round 3 continues the same corrective trajectory. The agent further reduces unroll-count to 4, raises slp-threshold to 8 to suppress unprofitable vectorization attempts, and inserts an additional GVN pass after unrolling. These changes again improve intermediate metrics: IPC rises to 2.49, and L1 data-cache misses decrease further. Nevertheless, execution time remains at 13.72 s, still about 27% slower than the baseline. The remaining optimization remarks indicate persistent LoadClobbered behavior and unresolved alias-analysis barriers, suggesting that the core bottleneck is now an alias-sensitive memory problem that the current edit space cannot effectively repair.

This failure case reveals three important properties of AutoPass. First, the system is interpretable: each round is linked to concrete compiler evidence and measured runtime effects rather than opaque search behavior. Second, the feedback loop is genuinely useful: from Round 1 to Round 3, AutoPass systematically improves IPC and lowers cache misses. Third, diagnosis does not guarantee recoverability. When the dominant bottleneck is governed by deep memory-system behavior and alias-sensitive load redundancy, the current edit space of pass reordering and parameter tuning may be insufficient to restore the baseline execution regime. This suggests that future improvements should incorporate stronger alias-analysis guidance or more explicit memory-aware optimization criteria.