AI-PROPELLER: Warehouse-Scale Interprocedural Code Layout Optimization with AlphaEvolve

Chaitanya Mamatha Ananda* **Rajiv Gupta** **Mircea Trofin** **Aiden Grossman**
Sriraman Tallam **Xinliang David Li** **Amir Yazdanbakhsh**
University of California, Riverside Google Google DeepMind
{cmama002,rajivg}@ucr.edu,{mtrofin,aidengrossman,tmsriram,davidxl,ayazdan}@google.com

Abstract

Post-link optimizers (PLOs) such as Propeller and BOLT have demonstrated that precise, profile-guided code layout can extract significant performance gains from heavily optimized binaries. However, these systems are currently restricted to intraprocedural techniques, leaving the global potential of interprocedural layout largely untapped. Interprocedural code layout is historically difficult due to a combinatorially intractable search space and complex call-return semantics that are challenging to model. Consequently, the performance potential of fine-grained interprocedural layout remains unproven in practice. AI-PROPELLER uses Magellan, an agentic workflow that evolves the compiler heuristic in Propeller into a fine-grained interprocedural optimizer and fine-tunes the resulting policy hyperparameters. To ensure high-fidelity, we move away from approximate static cost models and the agentic workflow generates multiple layout variants that are executed on actual hardware to measure real performance counters, providing a precise reward signal for the evolutionary loop. AI-PROPELLER has been evaluated on several benchmarks including large warehouse-scale applications and experiments show performance improvements of 0.23% to 1.6% optimized with state-of-the-art FDO and PLO which is significant for real-world binaries. This is the first time ever that large warehouse-scale applications in industrial settings have been optimized with fine-grained interprocedural code layout.

1 Introduction

Post-Link Optimizers (PLO) such as Propeller [30] and BOLT [26] have been used to significantly improve the performance of large warehouse-scale applications [13]. Recently, systems such as ACE [17] have demonstrated the potential for AI-driven optimization in warehouse-scale environments, though interprocedural layout optimizations remain an open challenge. At this scale, even marginal performance improvements [1, 10] have significant business value. Code Layout [27, 23] is an important optimization that has been particularly effective for such applications. These PLO optimizers use accurate profiles to reorder the basic blocks of hot functions, optimizing instruction cache and TLB utilization. Further, function splitting [11, 15] cleaves the frequently executed (hot) and rarely executed (cold) subsets of a function and places them in separate regions in memory to further improve TLB behavior. Collectively, these optimizations squeeze maximal performance by placing the hot working set of an application to reside in close proximity.

Code layout has been well studied and a number of prior works exist that reorder both basic blocks [2, 20, 22, 19, 16] and functions [27, 21, 8] for optimal utilization of instruction caches and TLBs. Early research [3] demonstrated the value of profile information in guiding code optimizations, while techniques such as trace scheduling [6] pioneered global code optimization to maximize

*Work done at Google.

AI Agents for Discovery in the Wild (AID-Wild), Workshop at ACM CAIS 2026.

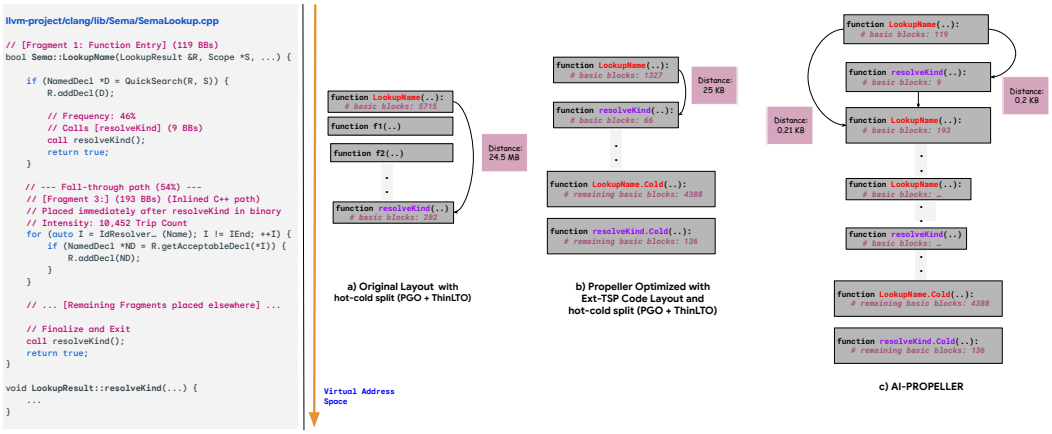

Figure 1: Motivating Example from a real-world application, *LLVM Clang* binary optimized with AI-PROPELLER

instruction-level parallelism. Particularly, the Ext-TSP heuristic [23] has proven to be very effective and is used by both Propeller and BOLT. However, these heuristics are applied intraprocedurally, one function at a time, and do not cross function boundaries with the exception of function splitting. While PLO frameworks such as Propeller possess the capabilities to reorder basic blocks across procedures arbitrarily (e.g. via basic block sections [30]), there are no existing code layout heuristics that exploit this opportunity to reorder blocks across functions for increased performance.

Interprocedural code layout, that is the ability to partition functions and reorder the partitions, is a significantly more challenging problem in comparison to intraprocedural layout, for two main reasons. First, the combinatorial complexity grows exponentially as the interprocedural search space is several orders of magnitude larger. For example, a warehouse-scale application we optimized has $\sim$ 1.4 million functions and the largest function has $\sim$ 20,000 basic blocks. Whereas, the entire application has $\sim$ 40 million basic blocks which adds several million orders of complexity over intraprocedural exploration. Scaling to an interprocedural algorithm is a significant engineering effort, which is hard to justify without demonstrable performance improvements. Second, the cost function to layout basic blocks across procedures is very different as call-return semantics cannot be violated even if two blocks from different procedures are placed adjacently, whereas, with intraprocedural layout the fall-through jumps between adjacent blocks are eliminated reducing the dynamic instruction count and also increasing cache utilization. To our knowledge, AI-PROPELLER is the first work to demonstrate clear performance potential on large warehouse-scale applications via interprocedural code layout.

AI-PROPELLER adopts the Magellan framework [4] to evolve code heuristics. Specifically, it leverages *AlphaEvolve* [24] to evolve existing heuristics, and a black-box optimizer, such as Vizier [9] to tune its numerical parameters. Both AlphaEvolve and Vizier rely on a *reward signal* to propose code edits and parameter updates, respectively. The reward is expected to be empirically obtained by applying the new heuristics to binaries of interest (or - for completeness - closely correlate with an empirical signal of this nature). More details in section 4.

In this context, we apply the framework to Propeller's [30] code layout optimization, which is currently formulated as an *Extended Traveling Salesman Problem* [23]. The key challenge that had to be overcome when applying Magellan was ensuring the reward signal is both *time – efficient* (to maintain practical training times) and *noise-free* (to prevent misleading AlphaEvolve or Vizier). To that end, we identified a small set of representative benchmarks (binaries comparable in working set size to the binaries of interest) that had a very low run-to-run variation in performance (lower than 0.05%). This paper makes the following contributions:

1. This is the first study to show performance improvements with interprocedural code layout optimizations on real-world applications, including large warehouse-scale binaries, resulting in performance improvements ranging from 0.23% to 1.6%. We want to stress that, while naively these improvements may seem small, they are both significant from a operational cost / business

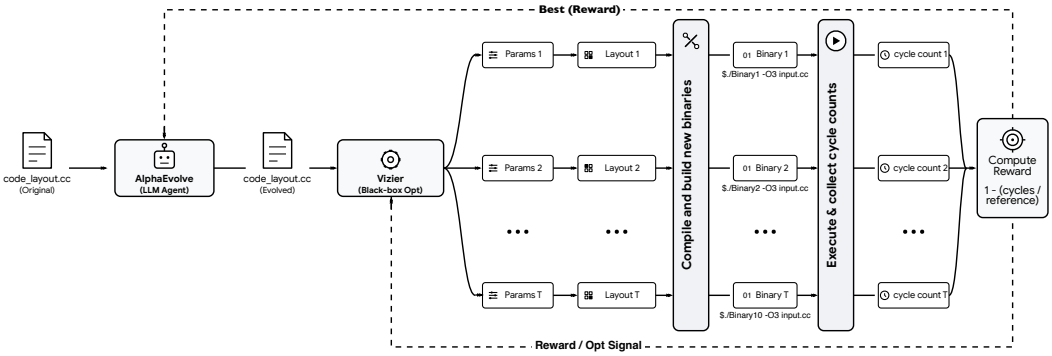

Figure 2: AI-PROPELLER Overview: Application of Magellan to Propeller

perspective, and very hard to achieve in industrial settings already employing the state of the art optimization techniques (profile-guided optimizations, link-time optimizations, and the current block-layout techniques)

2. We show that interprocedural block placement heuristics are achievable for large binaries in industrial settings without noticeably changing build characteristics (time and memory usage)

2 Interprocedural Code Layout in Propeller

Propeller [30] is an industry-strength Post-Link Optimizer that has been used to optimize large datacenter applications, particularly in environments with distributed build systems where build scalability is critical. As noted in the paper [30], the improved performance is primarily from intraprocedural code layout and function splitting [15]. Intraprocedural code layout optimizes each function's basic block layout for locality and reduces the number of dynamic taken-branches by placing two hot basic blocks that are successively executed adjacent to each other. Function splitting separates the hot and cold subsets of each function and clusters the hot and cold parts of all functions into separate regions. This reduces the working set size and improves locality and performance.

Propeller introduced *basic block sections* which allows arbitrary code layout of basic blocks in the virtual address space, even across functions. While Propeller has the capabilities for interprocedural code layout, it currently does not optimize binaries with interprocedural heuristics. As noted in the Propeller paper [30] in *Section 4.7*, while one isolated experiment with interprocedural layout did show a small performance improvement on one benchmark, generating such layouts took a very long time ($3X$ to $10X$) primarily due to the increased search space, and interprocedural layout has been labeled as future work.

AI-PROPELLER has been implemented in the Propeller framework. As it is already able to lay out code interprocedurally, the challenge was to find a heuristic that yields performance improvements above and beyond the state-of-the-art. We used *Magellan* to evolve the existing heuristic resulting in an improved interprocedural capable heuristic without affecting the build scalability. We have used AI-PROPELLER to optimize large and warehouse-scale applications without any perceivable degradation in build characteristics such as time and memory usage.

3 Motivating Example

To illustrate how interprocedural layout is able to further improve performance, we show a code snippet and the optimized code layout generated by AI-PROPELLER on a real-world application, *LLVM Clang* [18] compiler. In Figure 1, function `Sema::LookupName` calls `LookupResult::resolveKind` twice. In the original layout shown in Figure 1(a) for a PGO+ThinLTO [12] binary, these functions are placed arbitrarily and far apart, $\sim$ 24 MB, in the address space even though the calls are hot. With Propeller's intraprocedural layout and function reordering, these functions are split into their hot and cold subsets first and their hot subsets are placed close together, 25 KB apart, and this significantly improves locality resulting in better performance. The hot function size of `LookupName` is still 25 KB even after splitting, due to aggressive inlining.

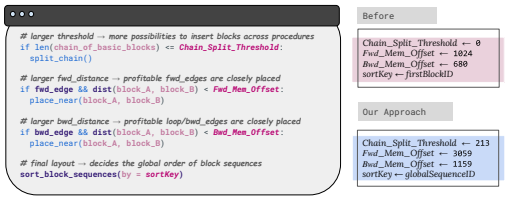

Figure 3: Discovered Policy from AI-PROPELLER for *LLVM Clang* [18]

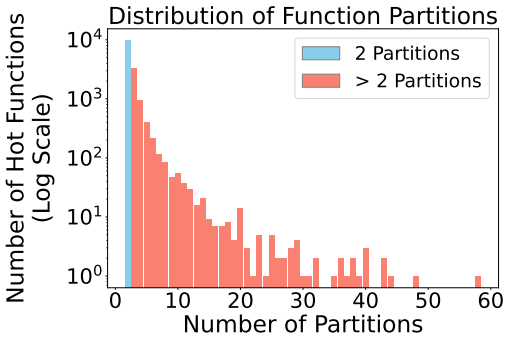

Figure 4: Distribution of Partitions in *LLVM Clang* [18] Binary Optimized with AI-PROPELLER

AI-PROPELLER takes this even further. It identifies the basic block subsets of the hot function partitions that are executed during calls to function `LookupResult::resolveKind` and places them together, 0.2 KB apart, significantly improving locality. Traditional hot-cold function splitting [15] only splits a function in two subsets, AI-PROPELLER can split a function into several subsets.

Looking at the final optimized binary of *LLVM Clang* [18] with AI-PROPELLER, as shown in Figure 4 we found that out of the $\sim$ 15000 hot functions that were considered as candidates for interprocedural layout, $\sim$ 5000 functions were partitioned in more than 2 ways with the maximum number of partitions being 57 and the mean number of partitions slightly larger than 3. This illustrates both the vastness of the search space with interprocedural layout and the effectiveness of AI-PROPELLER in finding layouts that improve performance.

4 Overview of AI-PROPELLER using Magellan

We will first describe Magellan [4]. Its use in our specific context is depicted in Figure 2. Next we discuss the improvements it made to an initially naive inter-procedural block placement heuristic.

Magellan. We use the Magellan approach [4] combining AI-guided heuristic discovery and black-box numerical optimization. This is a reward guided evolution mechanism. The process begins with an initial policy - a function - being presented to AlphaEvolve (AE), together with a LLM prompt. AE prompts the LLM with that prompt and the referenced function and obtains one or more edits on that initial function. Each edit consists of novel code additions / modifications / removals. Since the original function operated on internal Propeller data structures, AE may extract new features of code IR, for example, to consider in making decisions. As part of the Magellan methodology, the prompt includes a directive that any numerical parameters be exposed as compiler flags. For a policy thus proposed by AE, Magellan attempts the following:

1. Recompile Propeller by replacing the policy being evolved with the proposed policy, in the Propeller codebase. If this step fails (because the proposed edits cause compilation failure), Magellan feeds the build error messages back to AlphaEvolve (which includes them in the next prompt) and repeats the process;

2. For policies that can successfully be included in the Propeller codebase, Magellan identifies their parameters (the generated compiler flags) and starts a blackbox tuning session with e.g. Vizier [9].

Vizier attempts to tune those parameters based on the reward signal, which, as discussed, it obtains by running representative binaries in a noise-free environment. The best performing flag combination is presented back to Magellan, which records it and presents it back to AlphaEvolve together with the observed reward. The process then continues - AlphaEvolve samples out of its previously produced policies, mutates the samples, and generates new policies. The overall process has no intrinsic termination point.

4.1 Evolving the Code Layout Policy in Propeller

Interprocedural basic block reordering for a binary is governed by three factors. How sequences of basic blocks are split, where new blocks are inserted, and which edges are prioritized for profitability: *Chain Split Threshold*, *Forward Memory Offset*, and *Backward Memory Offset*. The search space has remained unexplored because manual tuning of these interdependent parameters is computationally intractable is of the order of *Billions* for large binaries. By leveraging *Magellan*, AI-PROPELLER identified a novel configuration that produces performance improving interprocedural layouts as shown in Figure 3. Specifically, AI-PROPELLER evolved the values of the three constants discussed above that determine the order of the layout. The specific improvements made to the policy are categorized below:

- **Optimal Split Points:** The current heuristic forms sequences of basic blocks while optimizing and determines split points in existing sequences to insert new blocks. The `chain_split_threshold` parameter determines the maximum sequence size above which sequences will not be considered for splitting. The existing Ext-TSP [23] had this at 0 avoiding any splitting possibility whereas AI-PROPELLER increased this to 213, splitting a lot more sequences.
- **Considering Edges Over Larger Distances:** The memory offset parameters, `Fwd_Mem_Offset` and `Bwd_Mem_Offset`, determine the maximum distance between two basic blocks for their connecting edge to be considered by the optimizer. The current heuristic [23] restricts this to 1024 bytes (forward) and 680 bytes (backward). This narrow window is sufficient for *intraprocedural* layout where hot paths are local (confined inside a single function). However, *interprocedural* layouts involve caller-callee and return relationships that frequently span over much larger memory distances. AI-PROPELLER expanded these limits to 3059 and 1159 bytes, respectively, allowing the optimizer to capture and minimize the distance of these previously ignored *long-range* edges.
- **Preserving the Interprocedural Layout:** While the numerical parameters determine how the various basic blocks are grouped into sequences, AI-PROPELLER also evolved the logic for the order in which these sequences are organized in the binary. Ext-TSP sorts the various resulting block sequences by their original block IDs. This is effective for *intraprocedural* layout, where the goal is simply to order blocks within a function. As the scope expands to *interprocedural* layout, AI-PROPELLER evolves the logic to support interprocedural optimization goals. To account for the local scope of block IDs, AI-PROPELLER generates a global layout index. By sorting according to this new index rather than function-specific block IDs, the system ensures that the final binary strictly preserves the intended order for basic blocks.

5 Experiments and Results

Training setup. AI-PROPELLER framework was trained on a representative subset of 100 modules derived from the *Ninja* [14, 5] build of *Clang* compiler. The training phase spanned 2.7 days, comprising 12 AlphaEvolve [24] iterations and a total of 1,200 Vizier [9] trials (distributed as 100 trials per AlphaEvolve iteration).

To maintain a high quality (noise-free) reward signal, all performance evaluations were conducted on Intel Skylake systems within a strictly controlled environment. Specifically, Turbo Boost, SMT and ASLR were disabled to minimize non-deterministic noise. Furthermore, the CPU frequency was locked at 75% of the peak frequency to eliminate thermal throttling and ensure consistent execution across trials. Each candidate binary was executed 10 times, yielding a run-to-run variance of 0.02% to 0.04%. This high stability enabled AI-PROPELLER to accurately attribute fine-grained performance gains to the proposed layout transformations.

5.1 Experimental Methodology

Benchmarks and baseline algorithms. Our evaluation covers a diverse set of applications, including *LLVM Clang* [18], *LevelDB* [7], *Redis* [29], and Search (a warehouse-scale application). AI-PROPELLER was evaluated and compared against the algorithms described in Table 1.

Configuration	Description
Baseline	The FDO [34] and ThinLTO [12] optimized binary with function layout disabled.
CDSort [28]	The state of the art function layout (reordering) algorithm applied to Baseline.
Ext-TSP [23]	The state-of-the-art algorithm for *Intraprocedural* basic block reordering applied to Baseline.
***AI*-PROPELLER**	Our Approach applied to Baseline.

Table 1: Summary of Evaluated Algorithms and Configurations

5.1.1 Policy Generalizability

We evaluated the generalizability of the policy discovered by AI-PROPELLER in Figure 3. While the policy was trained on a small test workload of only 100 modules, it generalized effectively to the full *LLVM Clang* build comprising $\sim 3,000$ modules. The same policy also performed well on *LevelDB* and *Redis* benchmarks. However, it did not generalize well to the warehouse-scale *Search* workload, yielding neutral performance. Consequently, we trained a separate policy for Search using its specific test workload and evaluated it accordingly.

5.2 Experimental Evaluation

To evaluate the effectiveness of AI-PROPELLER, we conducted experiments focusing on front-end bound [35] workloads, where instruction cache efficiency and branch prediction are critical. **Evaluation Environment.** All performance evaluations were executed on dedicated, bare-metal infrastructure (OVH Dedicated Server) using *LLVM*[1] with *ThinLTO* [12] enabled globally. Benchmarks were executed on an Intel Xeon Silver 4214R system running Arch Linux (Kernel 6.17.5). The hardware of the system consists of 48 cores and 96 GB of RAM. The cache hierarchy includes a 32 KB L1 instruction/data cache, a 1 MB private L2 cache per core, and a 16.5 MB shared L3 cache.

5.3 Analysis of Results

Our results demonstrate that AI-PROPELLER consistently outperforms traditional heuristics by discovering non-intuitive, interprocedural layouts. Across all the benchmarks, the performance gains are statistically significant with a p-value close to zero.

Reduction in execution times. The evaluation, Figure 5, of AI-PROPELLER demonstrates performance gains over the *Baseline* and prior state of the art across three distinct applications. Particularly, on a large real-world application like *clang*, the improvement is 1.6%. To put this in perspective, Propeller [30] alone improved this benchmark by 7.3% as noted in the paper. AI-PROPELLER further squeezes more than 20% of that improvement.

Frontend-Bound Stalls [35]. A common bottleneck across these workloads is high instruction fetch and decode stalls. AI-PROPELLER significantly mitigates these stalls, Figure 6 by improving code locality. In *Clang*, we observed a reduction in stalls compared to baseline from 45% to 39%. In *LevelDB* stalls decreased from 29% to 27%, while *Redis*, frontend-bound stalls dropped from 49% in the baseline to 34.2%, showing significant reduction in frontend stalls.

CPU pipeline slots spent retiring instructions. With reducing frontend stalls, top-down analysis [35] shows that AI-PROPELLER enables the processor to sustain a higher rate of retired instructions, Figure 7, which is a key indicator of useful work performed per cycle. AI-PROPELLER improved instruction retirement throughput in *LLVM Clang* [18] from 27% to 30%, in *LevelDB* [7] it improved from 51% to 54% and in *Redis* [29], we did not see an increase over the baseline, it remained comparable to Ext-TSP.

[1] Commit Hash: bf91a6226996439883654402 0def699e3f019b9b

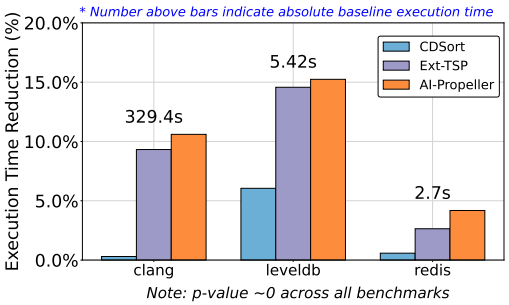

Figure 5: Execution Times of baseline and relative changes in execution times achieved by CDSort, Ext-TSP and AI-PROPELLER.

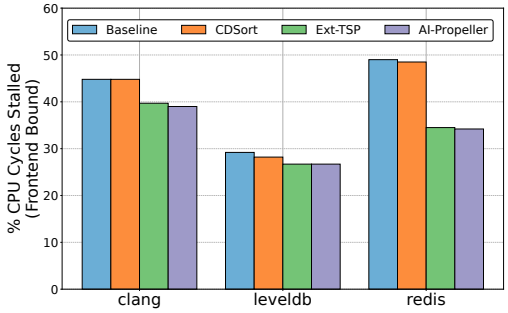

Figure 6: % of CPU cycles stalled on front-end events [35] across different algorithms, lower is better.

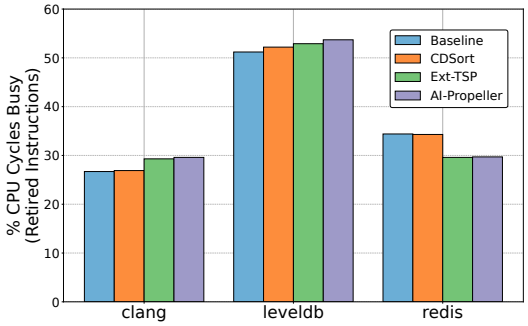

Figure 7: % of CPU pipeline slots spent in retiring instructions. Higher is better as it indicates more useful work done.

5.4 Warehouse Scale

To evaluate *AI*-PROPELLER, we applied it to a proprietary warehouse workload, *Search*. For the AlphaEvolve [24] generated policy from clang, Vizier [9] suggested a distinct set of hyper-parameters for *Search*. Unlike clang benchmark, for *Search*, *AI*-PROPELLER converged on to a more conservative chain split threshold of 53, while expanding memory offsets to 2074 (forward) and 1423 (backward) to account for the massive code footprint.

This discovered policy yielded a 0.23% improvement over the deployed Ext-TSP at production. While proprietary constraints limited the collection of frontend stall and instruction retirement metrics, this result demonstrates that *AI*-PROPELLER can identify non-obvious optimization opportunities even in hyper-optimized, warehouse-scale workloads.

5.5 Inter vs. Intra

We conducted an ablation study to isolate the performance gains we obtained on *clang* with AI-PROPELLER and identify the portion of the wins that were primarily due to interprocedural layout. We modified the optimal interprocedural layout generated by AI-PROPELLER and forced all basic blocks belonging to the same function to remain contiguous, effectively disabling interprocedural layout and preserving the intraprocedural orderings.

When these interprocedural transformations were reversed, the performance improvement in cycle counts dropped from 1.6% to just $\sim$ 0.3% over Ext-TSP. This significant delta indicates that while *AI*-PROPELLER does identify marginal intraprocedural improvements, the vast majority of the total speedup, approximately 80% is derived from its ability to interleave blocks across function boundaries. This confirms that global interprocedural reordering is the primary driver of the AI-PROPELLER framework's effectiveness.

Restricting AI-PROPELLER to train on intraprocedural layout of basic blocks for *LLVM Clang*, we observed only about $\sim 0.6\%$ improvement over Ext-TSP on *clang* reinforcing the need for interprocedural basic block reordering.

6 Related Work

Prior code layout optimization techniques fall into three broad categories: function-level reordering, basic-block reordering, and learning-based approaches. These techniques rely on static cost models that sometimes do not capture the nuances of the hardware leading to sub-optimal layout decisions.

Function-Level reordering. Procedure-level placements improved instruction cache and TLB locality via profile-guided positioning [27], temporal execution data, and procedure merging [21, 8]. Modern large-scale systems refine this using dynamic clustering heuristics, such as *HFSort* and *C3* [25], or cache-aware placement like *CDSort* [28]. However, these coarse-grained methods treat functions as atomic units. They cannot separate hot and cold regions or co-locate hot blocks across function boundaries—limitations our fine-grained approach directly resolves.

Basic block reordering.. Basic-block reordering addresses these atomic limitations [20], with early work exploring code replication for branch prediction [22]. The *Extended Traveling Salesperson based heuristic (Ext-TSP)* [23] is the state-of-the-art in intraprocedural code layout and utilized by modern post-link optimizers like *BOLT* [26] and *Propeller* [30]. While Propeller's basic block sections and Codestitcher [16] enable cross-function block stitching, these frameworks still rely on static objective functions. Further, CodeStitcher [16] relies on whole program link time optimizations that do not scale with large distributed build systems. Consequently, they struggle to effectively navigate the interprocedural search space.

Learning-Based optimization approaches. Machine learning is used to augment or replace static compiler heuristics. MLGO [33] applied neural network models to replace heuristics in an industrial setting. PIE [31, 32] utilizes LLMs to identify performance-improving source code edits for warehouse-scale applications. ACE [17] investigates AI-driven optimizations for these environments. Magellan [4] provides an agentic framework to evolve compiler passes using *AlphaEvolve*. AI-PROPELLER applies the latter approach to interprocedural code layout.

7 Conclusion

In this work, we present AI-PROPELLER, the first framework to successfully apply fine-grained, interprocedural code layout optimization to large binaries, including warehouse-scale applications. AI-PROPELLER has been built on top of the *Propeller* framework without compromising scalability. By applying the Magellan [4] approach of integrating AlphaEvolve and Vizier, our approach autonomously discovers novel layout policies, navigating an exponentially larger search space, and overcoming the limitations of traditional heuristics. We have built a highly stable execution environment that reduces run-to-run variance to under 0.05% that enables the evolutionary loop to learn directly from precise hardware performance counters rather than relying on approximate static cost models. Our evaluation demonstrates clear performance gains on real-world, front-end bound workloads. Most notably, *AI*-PROPELLER achieves a 0.23% execution improvement on a heavily optimized, warehouse-scale Search service compared to the state-of-the-art Ext-TSP baseline. Furthermore, it achieves a 1.6% improvement on a large real-world application, *LLVM clang* compiler. These results confirm that AI-driven agentic workflows can successfully extract additional performance from mature compiler infrastructures.

References

[1] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers," in *ISCA*, 2019.

[2] R. Bodík, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *PLDI*, 1997.

[3] P. P. Chang, S. A. Mahlke, and W.-m. W. Hwu, "Using profile information to assist classic code optimizations," *Softw. Pract. Exper.*, 1991.

[4] H. Chen, A. Novikov, N. Vũ, H. Alam, Z. Zhang, A. Grossman, M. Trofin, and A. Yazdanbakhsh, "Magellan: Autonomous discovery of novel compiler optimization heuristics with alphaevolve," in *arXiv*, 2026.

[5] Evan Martin, "Ninja," https://ninja-build.org/, version X.Y.Z, accessed May 8, 2026.

[6] Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, 1981.

[7] S. Ghemawat and J. Dean, *LevelDB*, 2011. [Online]. Available: https://github.com/google/leveldb

[8] N. Gloy, T. Blackwell, M. Smith, and B. Calder, "Procedure placement using temporal ordering information," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997.

[9] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. E. Karro, and D. Sculley, Eds., *Google Vizier: A Service for Black-Box Optimization*, 2017. [Online]. Available: http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization

[10] A. H. Hunter, C. Kennelly, D. Gove, P. Ranganathan, P. J. Turner, and T. J. Moseley, "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator," in *OSDI*, 2021.

[11] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *ISCA*, 1989.

[12] T. Johnson, M. Amini, and X. D. Li, "ThinLTO: scalable and incremental LTO," in *CGO*, 2017.

[13] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *ISCA*, 2015.

[14] Kitware, Inc., "Cmake," https://cmake.org/, version X.Y.Z, accessed May 8, 2026.

[15] S. Kumar, "[RFC] Machine Function Splitter - Split out cold blocks from machine functions using profile data," https://discourse.llvm.org/t/rfc-machine-function-splitter-split-out-cold-blocks-from-machine-functions-using-profile-data/56093, August 2020, lLVM Discussion Forums.

[16] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: inter-procedural basic block layout optimization," in *CC*, 2019.

[17] H. Lin, M. Maas, M. Roquemore, A. Hasanzadeh, F. Lewis, Y. Simonson, A. Shringi, H. Dai, P. Musau, T.-W. Yang, A. Y. Bakhsh, D. Altinbüken, F. Papa, M. N. Edmonds, A. Patil, D. Schwarz, S. Chandra, C. Kennelly, M. Hashemi, and P. Ranganathan, "ACE: An AI-Driven Code Efficiency Optimizer for Warehouse Scale Computers," in *OSDI*, 2026.

[18] LLVM Developer Group, *Clang: A C language family frontend for LLVM*, 2007. [Online]. Available: https://clang.llvm.org

[19] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, "Ispike: A post-link optimizer for the intel®itanium®architecture," in *CGO*, 2004.

[20] S. McFarling, "Program optimization for instruction caches," in *ASPLOS*, 1989.

[21] S. McFarling, "Procedure merging with instruction caches," in *PLDI*, 1991.

[22] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," 1995.

[23] A. Newell and S. Pupyrev, "Improved basic block reordering," *IEEE Transactions on Computers*, 2020.

[24] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "AlphaEvolve: A coding agent for scientific and algorithmic discovery," in *arXiv*, 2025.

[25] G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," in *CGO*, 2017.

[26] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: a practical binary optimizer for data centers and beyond," in *CGO*, 2019.

[27] K. Pettis and R. C. Hansen, "Profile guided code positioning," *SIGPLAN Not.*, 1990.

[28] T. L. Project, "llvm::codelayout::cdsortconfig struct reference," Doxygen API Documentation, 2025, https://llvm.org/doxygen/structllvm_1_1codelayout_1_1CDSortConfig.html.

[29] S. Sanfilippo, *Redis*, 2009. [Online]. Available: https://redis.io

[30] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, "Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications," in *ASPLOS*, 2023.

[31] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," in *ICLR (Spotlight)*, 2023.

[32] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Automated high-level code optimization for warehouse performance," *IEEE Micro*, 2025.

[33] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," 2021. [Online]. Available: https://arxiv.org/abs/2101.04808

[34] B. Wicht, R. A. Vitillo, D. Chen, and D. Levinthal, "Hardware counted profile-guided optimization," 2014. [Online]. Available: https://arxiv.org/abs/1411.6361

[35] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *ISPASS*, 2014.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction explicitly state the performance improvements achieved (0.23% to 1.6%) and clearly outline the framework's application to real-world warehouse-scale binaries

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification:

   - Policy Generalizability: We discuss limitations in Section 5.1.1, specifically regarding the policy generalizability. We acknowledge that while the discovered policy generalizes well to applications like LLVM Clang, LevelDB, and Redis, it did not perform optimally on the warehouse-scale "Search" workload without dedicated retraining.
   - Proprietary Constraints: As noted in Section 5.4, proprietary constraints limited our ability to collect detailed performance metrics (such as frontend stall and instruction retirement metrics) for the production "Search" workload, though we were still able to demonstrate performance improvements.
   - Computational/System Dependencies: The effectiveness of the AI-PROPELLER framework relies on a stable execution environment (e.g., locked CPU frequencies, disabled SMT/ASLR) to maintain a noise-free reward signal during training. We acknowledge that in less controlled industrial build environments, achieving similar noise-free signals may require additional engineering effort.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.

- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This work is empirical; we do not present new theoretical proofs or mathematical theorems.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provided detailed descriptions of the training/evaluation pipelines and the LLVM commit hash and the experimental setup in Section 5, allowing for replication of our methodology.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.

- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: Due to corporate proprietary constraints, we cannot release the full production binary dataset, but we provide all algorithmic implementation details and instructions for reproducing results on open-source benchmarks.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Data splits, hyperparameter selection, and optimizer settings are fully documented in Figure 3 and Section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.

- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We assess the statistical significance of our performance improvements by calculating p-values across experimental runs. As reported in Section 5 and Figure 5, we observe consistent improvements with a calculated p-value of approximately 0, confirming that the performance gains are statistically significant and not due to random variation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We disclosed the hardware used (e.g., specific hardware requirements) and the estimated compute time required for training in Section 5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

Justification: We have reviewed the NeurIPS Code of Ethics and ensured our research methodology conforms to these guidelines.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [N/A]

Justification: This is a foundational optimization technique for binary code; we do not anticipate any direct negative societal impacts from its use.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The research involves binary code optimization and does not involve high-risk assets like generative models or scraped public data.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.

- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All benchmarks and existing frameworks (e.g., LLVM, Propeller) used in our evaluation are cited and used according to their respective open-source licenses

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: We do not introduce new datasets or models that are being released as standalone assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: Our research does not involve human subjects or crowdsourcing.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This research involves the empirical performance evaluation of binary code optimization algorithms on hardware and does not involve human subjects, crowdsourcing, or personal user data.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: We employ an LLM as the core logic for the AI-PROPELLER agent to perform binary layout decisions, which is a central component of our method.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.