Formal Performance and Compile Time Guarantees for Compiler Optimization Heuristics

Nikil V. Shyamsunder
Department of Computer Science, Cornell University
nvs26@cornell.edu

***Abstract*—Modern optimizing compilers rely on heuristic search algorithms for NP-hard optimization problems, which can result in poor generated-code performance and long or unpredictable compile times. These are considered bugs by users, but verified compilers rarely reason beyond semantic preservation. We propose verifying performance and compile time properties of compiler passes. As a proof-of-concept, we formulate inline expansion using a cost model estimating instruction-cache performance. We mechanize this in Rocq, prove semantic preservation of the inlining transformation, and verify the algorithm's monotone improvement, convergence-time bound, and performance bounds for intermediate and final solutions.**

I. Introduction

Compilers are evaluated on correctness, the performance of generated code, and the compile time required to achieve it. Verified compilers have traditionally focused exclusively on correctness. CompCert, for example, leverages the technique of *bisimulation* proofs for semantic preservation: a source program step must correspond to a sequence of target steps with the same behavior, and vice versa [1]. Recent extensions use *quantitative* reasoning to guarantee resource availability on the target machine, such as stack-space or memory bounds [2], [3]. These rule out a large class of miscompilations, but do not reason about the optimality of the algorithms: how close a pass's output is to the best result in its search space.

In practice, excessively poor performance of generated code and long or unpredictable compile times are also treated as compiler bugs, revealing a mismatch between what verification guarantees and what users expect from a compiler. These bugs are often caused by global optimization passes where compilers rely on heuristics, such as register allocation and inline expansion in traditional software compilers [4], [5], [6]. A heuristic may find an excessively suboptimal solution or search too long for an acceptable solution. This problem is magnified in compilers for less conventional targets, such as ML frameworks and hardware toolchains, where the abundance of NP-hard problems such as tensor layout, hardware scheduling, placement, and routing means a poor search heuristic can miss high-performance transformations or incur long compile times [7], [8], [9], [10], [11]. In all of these settings, the compiler runs without formal guarantees that search will conclude quickly or find an effective solution.

For an entire production compiler, generated-code quality and compile time are emergent properties of many interacting passes and cost models, making them difficult to state as a single specification. Instead, we aim for guarantees about the particular passes where poor generated-code quality and runaway compile time are most likely to arise. Specifically, we focus on verifying two kinds of properties:

- **Performance bounds.** To rule out excessively poor generated code, we want a machine-checked bound on a pass's output relative to the best solution in its search space under some cost model. Even when no such bound is found, the formal reasoning is itself informative: it tends to pinpoint the input configurations under which the pass degrades, exposing the areas worth ameliorating.
- **Convergence time and progress bounds.** We want guarantees that a compiler optimization terminates in a bounded number of steps, and that varying the level of optimization through controls like `-O{N}` flags gives predictable compile time/code-performance tradeoffs.

We report our initial findings on a simple pass and cost model, as a first step toward providing production-ready algorithms that are provably correct and performant for critical passes.

II. Modeling Inline Expansion

As an initial example, we use *inline expansion*, which replaces a function call with a copy of the callee body. Inlining can remove call overhead and expose further optimizations, but may increase compile time, code size, and instruction-cache (i-cache) pressure. Inline expansion is at least as hard as the knapsack problem under a simple model where each candidate call site has an independent code size cost and expected benefit [12]. Standard knapsack approximation algorithms can therefore be used, but the independence assumption is strong since inlining one function changes the cost and profitability of later call sites. Heuristic approaches have been proposed using code size, i-cache behavior, optimization potential, call-graph ordering, and profile information [12], [13], [14], [15], [16], [4]. LLVM's inliner uses many of the above approaches, iterating over a worklist of call-graph components and processing call sites in bottom-up order. After a successful inlining, costs are recomputed and newly exposed call sites are appended.

For our prototype, we base our pass on LLVM's inliner but make certain simplifications. First, we model only i-cache performance costs. Formally, let $\mathcal{P}$ be the set of decisions the optimization must make; for inline expansion, each element $i \in \mathcal{P}$ is a candidate call site. Each call site chooses from $\mathcal{S}_i = \{\texttt{inline}, \texttt{preserve}\}$, and $\mathcal{S}$ is the space of choices for all call sites. Let $\mathcal{R}$ be the set of shared i-cache-line resources. For each call site i, there is an arbitrary mapping $\rho_i : \mathcal{S}_i \to 2^{\mathcal{R}}$

https://doi.org/

from each choice to the resources it uses. The cost function $\mathcal{C}$ encodes resource contention: performance is worse when more selected call sites share the same cache line, which we denote as *execution delay*. Thus, we model each resource r with an affine delay cost $c_r(x) = a_r x + b_r$, where x is the number of selected call sites using that resource. For an inlining solution profile $s \in \mathcal{S}$, a call site's delay D_i and the total execution delay for the program $\mathcal{C}$ are

$$D_i(s) = \sum_{r \in \rho_i(s_i)} c_r(|\{j : r \in \rho_j(s_j)\}|), \qquad \mathcal{C}(s) = \sum_{i \in \mathcal{P}} D_i(s).$$

This abstracts away many hardware details, but captures the first-order fact that inlined code increases code size and can raise pressure on hot i-cache lines.

Second, we assume all candidates are *leaf* call sites, so inlining a candidate does not introduce additional candidate calls into the worklist. Rather than a single iteration, we generalize by allowing multiple passes over the call sites in *any* ordering and run until reaching a fixed point. Specifically, each call site evaluates its execution delay under its two available choices and flips from *preserve* to *inline*, or vice versa, exactly when the other choice strictly reduces its delay; the procedure repeats these locally improving steps, recomputing costs at each step, until no candidate can improve. We now bound the worst-case convergence time to a fixed point, and the worst-case performance of the intermediate and final solutions.

III. Convergence & Performance Guarantees

We begin by noting that our inlining procedure is an instance of a well-studied mathematical structure from game theory. Treating call sites as players, their two choices as strategies, and their delays as payoffs (or in our case costs) gives a strategic game [17]; more specifically, our choices of $\mathcal{P}$, $\mathcal{S}$, and $\mathcal{C}$ form a *congestion game* [18] over the resource set $\mathcal{R}$. Our fixed-point iteration procedure is precisely *iterative best response* in game theory: repeatedly let one player switch to its best selfish choice while holding the others fixed [17].

Congestion games belong to the class of *exact potential games* [19], where a single global *potential function* reflects each individual player's cost change exactly [17]. The exact-potential property guarantees that every selfish, *local* inlining decision made via iterative best response strictly decreases a *global* progress measure. Consequently, the pass is guaranteed to converge to a fixed point, which is a *pure Nash equilibrium* (PNE) where no player can unilaterally improve. At a PNE, we can directly verify our performance bound by leveraging the *Price of Anarchy* (PoA). The PoA is the ratio of the maximum value of $\mathcal{C}$ at any PNE to the global minimum value of $\mathcal{C}$.

Because our specific formulation models i-cache contention using linear delays, it acts as an *affine* congestion game, for which Christodoulou and Koutsoupias bound the PoA at 2.5: our algorithm is at worst $2.5\times$ the global optimum [20]. While quite loose, this bound holds for any linear cost model over any cache geometry and code layout. This leaves a clear path for future work to tighten the bound by incorporating the specific details of the target architecture and program structure.

We also obtain an explicit compile time progress bound. We work with natural-valued costs, so each accepted best-response step decreases the underlying global potential function by at least one. If there are N candidate call sites, E cache-line resources, and every resource cost satisfies $a_r \le A$ and $b_r \le B$, then this progress measure is initially bounded by $E\,(AN^2 + BN)$. Therefore iterative best response converges within $E\,(AN^2 + BN)$ accepted steps. More generally, after any k accepted steps, the same argument shows that the remaining number of steps before convergence is at most $E\,(AN^2 + BN) - k$. We also prove $\mathcal{C}(s_k) \le 2(E(AN^2 + BN) - k)$, yielding a coarse intermediate performance bound. This gives a predictable compile time/code-performance trade-off, which `-O{N}` levels can expose: in a "fast compilation" mode like `-O1`, the compiler might halt the pass after a fixed k steps, and users can reason about the resulting performance.

IV. Mechanization in Rocq

In order to validate our methods, we mechanize our prototype in Rocq over a language inspired by CompCert's RTL, an intermediate representation where functions are control-flow graphs over elementary instructions that read and write pseudo-registers. We define the language's syntax and small-step operational semantics and define leaf inline expansion as a source-to-source relation parameterized by an arbitrary candidate call site. We then prove, under bisimulation, that the transformation defined by this relation is semantics-preserving.

Next, we formalize the iterative-best-response procedure that uses the affine cost model to decide *which* candidate call sites to inline. Once a subset of call sites has been chosen by the procedure, the compiler "performs" those inlinings using the expansion relation. We mechanically verify the PoA and convergence guarantees detailed in III. The definitions and proofs are approximately $4{,}000$ LOC with 0 admitted lemmas.

V. Conclusion & Future Work

We show that a verified compiler pass *can and should* certify performance bounds and convergence-time properties. Although this prototype is especially amenable to a strategic-game formulation and has strong theoretical guarantees due to its potential-game structure, we expect the same proof strategy to extend to richer cost models and other optimization passes, since game theory provides a natural language for relating local heuristic decisions to the global optimum and may allow us to reuse existing mechanized results [21], [22], [23]. Traditional approximation algorithm proof techniques may be required in certain settings. Three directions remain. First, we aim to instantiate the cost model with concrete, architecture-specific details, and to enrich the cost model toward production inliners such as LLVM's. Second, we plan to model *joint* cost models across multiple passes, which capture how one pass affects downstream behavior, enabling compositional guarantees that address phase ordering directly. Finally, we aim to extend the approach to more compelling optimization passes with unpredictable performance and compile time properties such as tensor layout or instruction scheduling.

REFERENCES

[1] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, p. 107–115, Jul. 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[2] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao, "End-to-end verification of stack-space bounds for c programs," *SIGPLAN Not.*, vol. 49, no. 6, p. 270–281, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594301

[3] M. O. Myreen, "The cakeml project's quest for ever stronger correctness theorems," in *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, pp. 1–1.

[4] T. Theodoridis, T. Grosser, and Z. Su, "Understanding and exploiting optimal function inlining," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 405–419.

[5] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[6] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.

[7] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.

[8] S. Sahni and A. Bhatt, "The complexity of design automation problems," in *Proceedings of the 17th Design Automation Conference*, 1980, pp. 402–411.

[9] A. Sllame and V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis," in *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, 2002, pp. 316–323.

[10] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on sdc formulation," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 433–438.

[11] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Resource-aware just-in-time opencl compiler for coarse-grained fpga overlays," *arXiv preprint arXiv:1705.02730*, 2017.

[12] R. W. Scheifler, "An analysis of inline substitution for a structured programming language," *Communications of the ACM*, vol. 20, no. 9, pp. 647–654, 1977.

[13] S. McFarling, "Procedure merging with instruction caches," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991, pp. 71–79.

[14] D. R. Chakrabarti and S.-M. Liu, "Inline analysis: Beyond selection heuristics," in *International Symposium on Code Generation and Optimization*, 2006, pp. 12–22.

[15] M. Serrano, "Inline expansion: When and how?" in *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 1997, pp. 143–157.

[16] P. Zhao and J. N. Amaral, "To inline or not to inline? enhanced inlining decisions," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 405–419.

[17] T. Roughgarden, *Twenty Lectures on Algorithmic Game Theory*. Cambridge: Cambridge University Press, 2016.

[18] R. W. Rosenthal, "A class of games possessing pure-strategy nash equilibria," *International Journal of Game Theory*, vol. 2, no. 1, pp. 65–67, 1973.

[19] D. Monderer and L. S. Shapley, "Potential games," *Games and Economic Behavior*, vol. 14, no. 1, pp. 124–143, 1996.

[20] G. Christodoulou and E. Koutsoupias, "The price of anarchy of finite congestion games," in *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, 2005, pp. 67–73.

[21] A. Bagnall, S. Merten, and G. Stewart, "A library for algorithmic game theory in ssreflect/coq," *Journal of Formalized Reasoning*, vol. 10, no. 1, pp. 67–95, 2017.

[22] N. Garg, "Econcslib: Ai-assisted lean formalization for economics & computation research," *arXiv preprint arXiv:2606.13306*, 2026.

[23] A. F. Ramos, D. B. Hulak, and R. J. de Queiroz, "Nash equilibria for finite games in isabelle/hol," *Arch. Formal Proofs*, vol. 2026, 2026.