Understanding and Exploiting Optimal Function Inlining

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
ETH Zurich
Switzerland

Tobias Grosser
tobias.grosser@ed.ac.uk
University of Edinburgh
United Kingdom

Zhenhong Su
zhendong.su@inf.ethz.ch
ETH Zurich
Switzerland

ABSTRACT
Inlining is a core transformation in optimizing compilers. It replaces a function call (call site) with the body of the called function (callee). It helps reduce function call overhead and binary size, and more importantly, enables other optimizations. The problem of inlining has been extensively studied, but it is far from being solved; predicting which inlining decisions are beneficial is nontrivial due to interactions with the rest of the compiler pipeline. Previous work has mainly focused on designing heuristics for better inlining decisions and has not investigated optimal inlining, i.e., exhaustively finding the optimal inlining decisions. Optimal inlining is necessary for identifying and exploiting missed opportunities and evaluating the state of the art. This paper fills this gap through an extensive empirical analysis of optimal inlining using the SPEC2017 benchmark suite. Our novel formulation drastically reduces the inlining search space size (from $2^{349}$ down to $2^{52}$) and allows us to exhaustively evaluate all inlining choices on 1,135 SPEC2017 files. We show a significant gap between the state-of-the-art strategy in LLVM and optimal inlining when optimizing for binary size, an important, deterministic metric independent of workload (in contrast to performance, another important metric). Inspired by our analysis, we introduce a simple, effective autotuning strategy for inlining that outperforms the state of the art by 7% on average (and up to 28%) on SPEC2017, 15% on the source code of LLVM itself, and 10% on the source code of SQLite. This work highlights the importance of exploring optimal inlining by providing new, actionable insight and an effective autotuning strategy that is of practical utility.

1 INTRODUCTION
Function inlining (aka inlining expansion) is one of the fundamental compiler transformations. It depicts size improvement due to inlining for the SPEC2017 benchmark suite (not including the Fortran-only benchmarks) for LLVM’s -Os optimization level; in the best case, enabling inlining results in up to 3x size improvement. For example, for the benchmark “leela”, the resulting binary size with inlining enabled is 30% of that with inlining disabled when compiling at LLVM’s -Os.

Making good inlining decisions is difficult; good choices depend not only on other inlining choices, but also on the rest of the optimization pipeline. For example, inlining may enable dead code elimination or lead to code size bloat. An inlining heuristic must balance enabling further compiler optimizations and size increase. The general inlining problem is as hard as the NP-complete knapsack problem [22]. Thus, many inlining heuristics have been proposed [7, 20, 23, 24, 28]; they consider various program features, e.g., the number of instructions, the call-site context, or an estimation of the compile-time impact. Profiling information in a JIT-compiled environment or through Profile Guided Optimizations (PGO) can also drive inlining heuristics [4, 21]. For example, very cold functions (i.e., functions that are unlikely executed) are not inlined when optimizing for performance. One proposed method for sidestepping the difficulty of predicting the cascading effects of inlining are trials [11]: the compiler tentatively inlines a function to evaluate its effectiveness and backtracks whenever it makes sense.

Despite the abundant work on inlining, there has been no systematic study to investigate optimal inlining (i.e., finding the best inlining choices) and evaluate the state of the art against it; only empirical analyses of inlining strategies exist [2, 3, 10, 12, 16, 25]. Insights into optimal inlining not only help understand how well
the state of the art performs, but also help develop more effective inlining strategies. Thus, the key objective of this work is to conduct the first systematic, extensive empirical analysis of optimal inlining.

As the first piece of foundational empirical work on understanding optimal inlining, we focus on binary size, which is a deterministic metric that does not depend on workload selection, while performance does. We believe that it is critical to first establish complete and principled foundations for reasoning about inlining choices before extending toward the practically more complex target of runtime optimization. In addition, optimizing for size is important in situations such as web applications [13], as well as modern mobile apps whose code size can reach over 100MB: "Reduced application size is critical not only for the end-user experience but also for vendor’s download size limitations. Moreover, download size restrictions may impact revenues for critical businesses."[6].

One challenge for studying optimal inlining at a realistic scale is the sheer size of the potential inlining choices—the straightforward search space includes $2^n$ inlining alternatives, where $n$ is the number of inlinable call sites in a program. To tackle this challenge, we propose a novel, alternative search space formulation that takes advantage of a call graph’s connectivity and leads to significantly fewer evaluations of inlining configurations when searching for the optimal. Indeed, for our study of optimal inlining on the SPEC2017 benchmark suite, our formulation reduces the search space from $3^{495}$ to $24$, which allows us to compute and analyze optimal inlining on 1,135 SPEC2017 C/C++ source files.

To evaluate how well the state-of-the-art inlining strategies perform, we use our inlining search space formulation to find the optimal configurations w.r.t. binary size on these 1,135 SPEC2017 files. We compare the state-of-the-art inlining strategy in LLVM[2] with optimal inlining. Our results show a clear gap, thus suggesting opportunities for designing better inlining strategies (Section 4).

We examine and characterize the optimal inlining configurations, and observe a prevalent local independence property among connected call edges in call graphs for the SPEC2017 files. This insight motivates us to introduce a new, simple autotuning strategy for inlining that exploits this property. Results show that our autotuning strategy outperforms LLVM by 7% on average across all SPEC2017, up to 4x on individual files, and up to 28% on individual benchmarks (Section 5.2.2). We also apply our autotuning on LLVM’s own codebase and SQLite; we obtain a 15% improvement over LLVM on the former and 10% on the latter (Section 5.2.3), highlighting the practical utility of our autotuner for rapidly reducing the program size of relevant applications, e.g., by utilizing "compilation farms".

This paper initiates the study of optimal inlining and highlights its importance; it makes the following contributions:

- The first extensive systematic study of optimal inlining on the SPEC2017 benchmark suite by comparing the inlining heuristics of a state-of-the-art optimizing compiler (LLVM) against optimal inlining for program size (Section 4); and
- A simple, effective inlining autotuning strategy that exploits insights from the optimal inlining study which, when evaluated for program size, leads to significant improvement over the state of the art on SPEC2017, SQLite, and the codebase of LLVM itself with an overall ~6,000,000 LoC (Section 5).

The rest of the paper is organized as follows. We first present necessary background (Section 2) and introduce our formulation of the inlining search space (Section 3). We then present our analysis of optimal inlining on SPEC2017 (Section 4) for program size. Next, we introduce our autotuning strategy (Section 5.1) and demonstrate its effectiveness on SPEC2017, SQLite, and LLVM (Section 5.2). We then discuss the impact of our work (Section 6). Finally, we discuss related work (Section 7) and conclude (Section 8).

2 BACKGROUND

This section gives the needed background on function inlining (also known as inlining expansion). We define several relevant terminologies and provide examples for illustration.

Inlining is the process of replacing a function call with the callee’s body. Consider the code fragment in Listing 1 and the corresponding generated assembly fragments for in Listing 2 and Listing 3: inlining bar (not shown in the assembly listings) extends the analysis scope of the compiler; it can determine that (bar(1) == 1) is always satisfied in the first loop iteration, therefore the generated code just checks if the input argument $n$ is positive. The non-inlined version (Listing 3) includes all the original loop logic since the compiler cannot determine that it is unnecessary.

```
int bar(int a) {
    return a + a;
}

int foo(int n) {
    for (int i = 0; i < n; ++i) {
        if (bar(1) == 1) return 0;
    }
    return 1;
}
```

Listing 1: Source Code

```
pushq rbp
pushq r14
pushq rbx
movl $1, r14d
testl rdi, redi
jle .LBB1_5
movl rdi, rbp
xorl rbp, rbp
.LBB1_3:
movl rbp, redi
callq bar
cmpl rax, rax
je .LBB1_4
addl $1, rax
cmpl rax, rax
je .LBB1_3
jmp .LBB1_5
.LBB1_4:
xorl r14d, r14d
.LBB1_5:
```

Listing 2: foo inlined

```
pushq rbp
pushq r14
pushq rbp
```

Listing 3: foo not inlined

---

1Not all functions can be inlined, e.g., an inliner may be unable to handle recursive functions, or a callee that is defined in a different translation unit.

2Using the optimization level "-O3", which is designed for size optimization.

We refer to the former as \( \text{(Section 3.1)} \) to illustrate the challenges and motivate our recursively

3 FORMULATE THE INLINING SEARCH SPACE

This section presents our novel formulation of the inlining search space. We first discuss the straightforward exponential search space (Section 3.1) to illustrate the challenges and motivate our recursively partitioned search space (Section 3.2).

Inlining works on those functions that can be inlined, \( \text{i.e., the inlinable functions} \). Not all functions are inlinable because, for example, an inliner might not be able to handle recursive functions, or the callee’s body might be unavailable.

Inlining operates on call graphs. A program’s call graph consists of functions (the nodes) and function calls (the edges). An inlining heuristic decides for each inlining candidate, \( \text{i.e., function call} \), if it should be inlined. We represent these two choices on a call graph with the following transformations:

- **Inlining an edge (call):** The two adjacent nodes (functions) are “merged”. If the callee is invoked in additional call sites, it is cloned before merging to preserve it for these call sites.
- **Not-inlining an edge (call):** The edge is marked as “no-inline”.

Note that the corresponding call still exists in the program, but is no longer considered for inlining.

We refer to the former as **inlining a candidate** and the latter as **not-inlining a candidate**. For example, the \( \text{A} \rightarrow \text{B} \) call in Figure 2(a) is an inlining candidate. If it is not-inlined as shown in Figure 2(b), the corresponding edge is simply preserved (marked by the dashed edge). Otherwise, if it is inlined as shown in Figure 2(c), the two nodes are merged; the edge \( \text{AB} \rightarrow \text{C} \) corresponds to the original \( \text{B} \rightarrow \text{C} \) call. Note that a clone of \( \text{B} \) is merged to \( \text{A} \) since there is another caller, \( \text{D} \).

We define an **inlining configuration** as the assignment of labels (\text{inline, no-inline}) to all inlining candidates. The inlining configuration of the call graph in Figure 2(a) would be \((\text{A} \rightarrow \text{B}) : \text{inline}, (\text{B} \rightarrow \text{C}) : \text{no-inline}, (\text{D} \rightarrow \text{B}) : \text{no-inline})\).

Inlining may introduce multiple copies of the same call. In the inlining graph of Figure 2(e), edges \( \text{B} \rightarrow \text{C} \) and \( \text{AB} \rightarrow \text{C} \) correspond to the same (original) call. Depending on the inlining strategy and the inliner’s capabilities, these edges may be treated independently, \( \text{i.e., one may be inlined and the other not, or they may be coupled. In this work, we assume the latter, but supporting the former requires a straightforward extension.} \)

Figure 2: Inlining example: (a) initial call graph, the \( \text{A} \rightarrow \text{B} \) (blue edge) is an inlining candidate; (b) the call was not inlined, illustrated by the dashed edge; (c) the call was inlined, \( \text{A} \) and \( \text{B} \) were merged, and an additional edge corresponding to the \( \text{(B, C)} \) call was inserted; \( \text{B} \) is not removed since it has one additional caller, \( \text{i.e., D} \).
3.2 Recursively Partitioned Search Space

Partitioning a call graph’s inlining search space is not limited to its connected components. Two observations enable this:

1. Connected components are independent w.r.t. inlining.
2. Not inlining a bridge\(^4\) is identical to deleting it w.r.t. inlining: additional independent components are created.

The second observation holds for the following reason. Each inlined call can potentially extend the scope of compiler transformations. Inlining multiple adjacent calls increases the optimization scope even further, which leads to the need for exhaustive search. However, the optimization scope is not expanded across non-inlined calls. Inlining a callgraph bridge, \(B\), that connects two callgraph components, \(C_1\) and \(C_2\), is the only way to combine their optimization scopes. Thus, \(C_1\) and \(C_2\) are independent w.r.t. inlining if \(B\) is not inlined, and they can be independently searched.

Partitioning a call graph across bridges leads to a potentially smaller search space. Given a call graph \(G\) with \(N\) edges, and a bridge \(B\) connecting components \(C_1\) and \(C_2\) with respectively \(N_1\) and \(N_2\) (\(N_1 + N_2 = N - 1\)) edges: (1) the naive search space size is \(2^N\); (2) the partitioned one is \((2^N_1 + 2^N_2 - 1) + 2^{{N-1}}\). The first parenthesized term corresponds to the search space size of the two components if \(B\) is not inlined (+1 for evaluating the combined result), and \(2^{{N-1}}\) corresponds to the search space size if \(B\) is inlined.

The example in Figure 5a demonstrates how partitioning across callgraph bridges can reduce the search space size: \(K \rightarrow L\) is a bridge between \([F, G, K]\) and \([L, H, I]\); if it is not inlined, the remaining decisions do not have any “inter-component” effects, e.g., inlining \(G \rightarrow K\) does not affect the transformations applied on \([L, H, I]\) in Figure 5b. The inlining search space of the Figure 5a call graph can be partitioned based on this observation:

- If \(K \rightarrow L\) is not inlined (Figure 5b), the two resulting components can be independently explored. Each of them has 2 edges, thus \(2 \times 2^2 = 8\) inlining configurations must be evaluated. This results in two partial inlining configurations: \(\{F \rightarrow G\} \Rightarrow choice_0\), \(\{G \rightarrow K\} \Rightarrow choice_1\) and \(\{L \rightarrow H\} \Rightarrow choice_2\), \(\{H \rightarrow I\} \Rightarrow choice_3\). To combine these into a complete inlining configuration (which includes \(K \rightarrow L\) as no inlining) one additional program size evaluation (compilation) is necessary.
- If \(K \rightarrow L\) is inlined (Figure 5c), the resulting call graph has 4 edges, therefore its search space size is \(2^4 = 16\).
- The combined size is \((2^2 + 2^2 + 1) + 2^3 = 25\), which is smaller than \(2^5 = 32\) under the naive formulation.

This partitioning scheme can be applied recursively to explore all inlining configurations. New bridges are created as the callgraph is dynamically updated by removing non-inlined edges or merging nodes across inlined ones. These newly-formed bridges are used to further reduce the search space size. We use the name independent inlining components for the components that are formed by ignoring no-inlining edges. We call the resulting search space recursively partitioned search space.

One way to visualize the search space of this approach is the inlining tree. The first layers of the Figure 5 example’s inlining tree are shown in Figure 6. Each tree node contains the set of (potentially merged via inlining) call graph nodes. The root of Figure 6 includes all nodes of Figure 5a. Each edge of the tree assigns a label, either inline or no-inline, to a call graph edge. The edges attached to the root of Figure 6 assign labels to \(K \rightarrow L\). The left subtree corresponds to Figure 5c, where nodes \(K\) and \(L\) are merged. The right subtree corresponds to Figure 5b and the two independent components are shown in the rectangular node. Each path from the root to a leaf corresponds to an inlining configuration. Paths that cross rectangular nodes are missing labels for edges in other independent components; they are partial inlining configurations. Three kinds of tree nodes exist:

- InliningTreeLeafs (not shown in Figure 6): correspond to inlining configurations.
- InliningTreeBinaryNodes (elliptical nodes in Figure 6) contain an independent inlining component; the edges connecting it with its children assign opposite labels to the same edge.
- InliningTreeComponentsNodes (rectangular nodes in Figure 6) contain multiple InliningTreeBinaryNodes; one for each independent inlining component.

An inlining tree can be used to exhaustively search for the optimal inlining configuration in the recursively partitioned space. The search space size is the number of InliningTreeLeafs plus the number of InliningTreeComponentsNodes in the tree: each leaf corresponds to a (partial) inlining configuration that must be evaluated, and each set of independent inlining components requires an extra evaluation to combine the best child configurations.

Figure 5: The inlining search space in Figure 5a can be partitioned on edge \(K \rightarrow L\). The reduced search space size is \(2^4 + 2^2 + 2^2 + 1 = 25\), while the naïve non-partitioned one is \(2^5 = 32\).

Figure 6: Inlining tree for the call graph of Figure 5. Each subtree corresponds to one inlining decision. Sibling subtrees assign different inlining labels to the same edge. The rectangular node contains the two independent components which were formed by not inlining the \((K \rightarrow L)\) call. Both of them can be explored independently.
The optimal inlining configuration is found by recursively propagating the best configurations from the leaves up to the root (Algorithm 1). All leaves are evaluated by compiling the target program with the corresponding inlining configurations and measuring the resulting binary sizes. InliningTreeBinaryNodes select the best configuration from their children. InliningTreeComponentsNodes combine the configurations of their children (by simply appending them since they are independent), and the new inlining configuration is evaluated and propagated. In the end, one configuration, the optimal, will reach the root. This evaluation scheme is embarrassingly parallel and most evaluations can be executed concurrently in different cores/machines.

An inlining tree is constructed from a call graph by recursively assigning inlining labels to the graph’s edges (Algorithm 2). An InliningTreeBinaryNode is used for single independent inlining components. At each such node a partition edge must be selected and two subtrees are attached to the node: one with the edge inlined and one with it not inlined. If multiple independent inlining components exist, an InliningTreeComponentsNode is used; the tree construction proceeds in each of the node’s children. If there are no unlabeled edges, an InliningTreeLeaf is attached. Recursive calls are treated in the same way as regular calls. It is the inliner’s responsibility to correctly inline them (e.g., to a certain depth).

The partition edge selection is important as inlining trees are not unique. For example, if the edges \( F \rightarrow G \), \( G \rightarrow K \), \ldots, are selected sequentially in Figure 5a, no InliningTreeComponentsNode will be introduced, and there will not be any search space size reduction. It is important to prioritize bridges such that many independent components arise. In our implementation we use the following heuristic (SelectPartitionEdge in Algorithm 2):

- If the call graph contains bridges, then the bridge adjacent to the least eccentric vertex (among the vertices adjacent to bridges) is selected, i.e., the vertex with the least maximum distance from any other vertex. This prioritizes central bridges.
- Otherwise, among the edges adjacent to the node with the highest out-degree, the one adjacent to the node with the least in-degree is chosen. This heuristic tries to balance two metrics: (1) the reduction of high out degrees since they can block partitioning, and (2) creating as many bridges as possible by removing edges adjacent to low in-degree nodes.

Using a heuristic for edge-selection does not affect the optimality of the tree’s evaluation. However, it does affect the number of different configurations that will be explored, i.e., a bad selection heuristic can lead to exploring all 2^n configurations, while a good one may lead to potentially orders of magnitude fewer.

The presence of recursive functions can result in an infinitely large search space. We can bound the number of possible configurations by setting a limit to recursive inlining. Without loss of generality, we inline recursive functions at most once.

4 ANALYZE OPTIMAL INLINING ON SPEC2017

This section presents our investigation into optimal inlining on the SPEC2017 benchmark suite. Exhaustive search for optimal inlining is necessary for evaluating state-of-the-art inlining heuristics and identifying missed inlining opportunities. The search space reduction of our recursively partitioned search space makes such empirical studies feasible on realistic benchmarks whose callgraph maximum degrees are reasonably small. We demonstrate this by evaluating a subset of the SPEC2017 benchmarks. Out of the 3,258 files, 746 are trivial w.r.t. inlining—they require no inlining decisions. We focus on the remaining 2,512 files.\(^5\)

\(^5\)We perform our analysis on individual source files and not at the whole program level due to the compilation model that C and C++ compilers use: calls across source files are resolved at link time and cannot be inlined.
Table 1: Search space size reduction on a subset of SPEC2017 (1,186 call graphs with recursive space size up to $2^{20}$). The total reduction is approximately $2^{349} \rightarrow 2^{25.2}$. The recursively partitioned space enables exploring larger call graphs and it significantly reduces the cost of exploring smaller ones.

<table>
<thead>
<tr>
<th>Search Space</th>
<th>Per file size percentiles ($\log_2$)</th>
<th>Geometric Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>Median: 8 75th: 18 95th: 38 Max: 349</td>
<td>7.57</td>
</tr>
<tr>
<td>recursive</td>
<td>Median: 6.2 75th: 10.9 95th: 17.4 Max: 19.9</td>
<td>5.42</td>
</tr>
</tbody>
</table>

4.1 Search Space Reduction

We first demonstrate the search space reduction magnitude. We select all SPEC2017 files whose recursively partitioned search space sizes are up to $2^{20}$ (1,186 files), and we compare them with the naive space sizes (Table 1). The reduction ranges from a few percent to several orders of magnitude. The largest one is $2^{349} \rightarrow 2^{10}$. The total search space reduction is approximately $2^{349} \rightarrow 2^{25}$ (or $2^{243} \rightarrow 2^{25}$ if we exclude the largest call graph). Our recursive search space formulation enables:

- Exhaustively exploring larger call graphs, even ones with an extreme number of naive inlining configurations.
- Exhaustively exploring significantly fewer inlining configurations in smaller call graphs.

4.2 Roofline Analysis vs. LLVM

We want to understand the gap between the state-of-the-art and the optimal in the context of function inlining for binary size. The optimal inlining configuration yields the optimal binary size, however, multiple inlining configurations that achieve optimality may exist; any of them is is sufficient for our purpose.

Using our reduced search space we evaluate the inlining heuristic of a modern optimizing compiler, LLVM, against the optimal. We exhaustively evaluate all SPEC2017 inlining trees with search space size up to $2^{18}$ ($n = 1$, 135) and compare the resulting text section size with LLVM’s output (Figure 7). In 46% of the cases, LLVM’s inlining heuristic can find the optimal; however, it fails to do so in the rest: the median size overhead in the non-optimal cases is 2.37%, 16% of the cases have an overhead of at least 5%, 8.5% have an overhead of at least 10%, and the maximum is 281%. LLVM’s inlining heuristic performs very well in the majority of the cases, but there is still room for improvement.

Our dataset contains 15,005 inlining decisions. LLVM agrees with optimal inlining in 72.7% of them (Table 2). In 23.7%, LLVM’s heuristic was too aggressive and inlined too many calls. On the other hand, it was too conservative in 3.6% of them. In total, 7,613 (50.7%) of the calls were not inlined and 7,392 (49.3%) inlined in the optimal configurations; LLVM did not inline 4,594 (30.6%) and inlined 10,411 (69.4%) calls. LLVM is too eager to inline calls, this can also be seen in a few sample call graphs (Figure 8) where this eagerness results in a significant size increase.

We also examine the length of the optimal inlined call chains; an inlined call chain is a call graph path whose edges have been inlined. The most prevalent call chain length is 1 (Figure 9); there are very few long inlined call chains. This implies that good inlining choices for binary size can be largely taken by only considering a local scope. We use this insight to design a simple but effective autotuning strategy for inlining.
5 LOCAL INLINING AUTOTUNER FOR SIZE

As we argue in Section 3.2, no inline’d edges partition the search space. This enables independent search in the resulting independent inlining components. We derive the following insights from our optimal inlining study (Section 4):

- A large percentage of edges is not inlined (Table 2).
- Shorter inlined call chains are more prevalent (Figure 9).

There are two special cases of the above insights.

(1) Optimal configurations without any inline’d edges.

(2) Optimal configurations with inlined chains of up to length 1.

In both cases the result would be the optimal configuration: no inline’d edges.

We evaluate our proposed inlining autotuner for program size (Section 5) on the SPEC2017 benchmark suite; we compare against LLVM’s inlining decisions and the optimal configurations whenever they are available (Section 5.2.1 and Section 5.2.2). We also present a case study on real-world systems software: LLVM’s and SQLite’s source code (Section 5.2.3). The research questions (RQ’s) that we aim to answer are:

- RQ1: How effective is local autotuning versus LLVM’s inlining strategy on SPEC2017? (Section 5.2.1)
- RQ2: How effective is round-based autotuning? (Section 5.2.2)
- RQ3: Can local autotuning be effectively applied to real-world software? (Section 5.2.3)

5.1 The Autotuner

We take advantage of the above insights to design a simple, embarrassingly parallel, and effective inlining autotuner (Algorithm 3).

We run all benchmarks on an AMD Ryzen Threadripper 3990X based system running Ubuntu 18.04. We based our work on LLVM version 11.0.3:\(^{6}\) the only modifications are on InliningAdvisor\(^ {7}\).

5.2 Evaluation

Theautotuner can reduce the size of 14 out of the 20 SPEC2017 benchmarks by up to 27.6%; the 5 regressions can be eliminated by initializing the tuning session with LLVM’s own inlining decisions. The total reduction across all SPEC2017 files in the latter case is 4.86%; by combining the results of clean slate and LLVM-initialized autotuning the total reduction drops to 6.05%. We can find 921 (81%) optimal inlining configurations out of the 1,135 exhaustively analyzed files (LLVM finds only 526). Four rounds of tuning can further reduce sizes of individual benchmarks by up to an additional 10% and the total size is reduced by 7.05%. Also, our autotuner can reduce the size of LLVM by up to 15.21% and of SQLite by up to 10.25%.

We use all benchmarks on a AMD Ryzen Threadripper 3990X based system running Ubuntu 18.04. We based our work on LLVM version 11.0.3:\(^ {6}\) the only modifications are on InliningAdvisor\(^ {7}\).

5.2.1 Autotuning on SPEC2017

We evaluate our proposed inlining autotuner for program size (Section 5) on the SPEC2017 benchmark suite; we compare against LLVM’s inlining decisions and the optimal configurations whenever they are available (Section 5.2.1 and Section 5.2.2). We also present a case study on real-world systems software: LLVM’s and SQLite’s source code (Section 5.2.3). The research questions (RQ’s) that we aim to answer are:

- RQ1: How effective is local autotuning versus LLVM’s inlining strategy on SPEC2017? (Section 5.2.1)
- RQ2: How effective is round-based autotuning? (Section 5.2.2)
- RQ3: Can local autotuning be effectively applied to real-world software? (Section 5.2.3)

Summary Results: Our autotuner can reduce the size of 14 out of the 20 SPEC2017 benchmarks by up to 27.6%; the 5 regressions can be eliminated by initializing the tuning session with LLVM’s own inlining decisions. The total reduction across all SPEC2017 files in the latter case is 4.86%; by combining the results of clean slate and LLVM-initialized autotuning the total reduction drops to 6.05%. We can find 921 (81%) optimal inlining configurations out of the 1,135 exhaustively analyzed files (LLVM finds only 526). Four rounds of tuning can further reduce sizes of individual benchmarks by up to an additional 10% and the total size is reduced by 7.05%. Also, our autotuner can reduce the size of LLVM by up to 15.21% and of SQLite by up to 10.25%.

We use all benchmarks on a AMD Ryzen Threadripper 3990X based system running Ubuntu 18.04. We based our work on LLVM version 11.0.3:\(^ {6}\) the only modifications are on InliningAdvisor\(^ {7}\).

5.2.2 Evaluation on Real-World Systems Software

We evaluate our proposed inlining autotuner for program size (Section 5) on the SPEC2017 benchmark suite; we compare against LLVM’s inlining decisions and the optimal configurations whenever they are available (Section 5.2.1 and Section 5.2.2). We also present a case study on real-world systems software: LLVM’s and SQLite’s source code (Section 5.2.3). The research questions (RQ’s) that we aim to answer are:

- RQ1: How effective is local autotuning versus LLVM’s inlining strategy on SPEC2017? (Section 5.2.1)
- RQ2: How effective is round-based autotuning? (Section 5.2.2)
- RQ3: Can local autotuning be effectively applied to real-world software? (Section 5.2.3)

Summary Results: Our autotuner can reduce the size of 14 out of the 20 SPEC2017 benchmarks by up to 27.6%; the 5 regressions can be eliminated by initializing the tuning session with LLVM’s own inlining decisions. The total reduction across all SPEC2017 files in the latter case is 4.86%; by combining the results of clean slate and LLVM-initialized autotuning the total reduction drops to 6.05%. We can find 921 (81%) optimal inlining configurations out of the 1,135 exhaustively analyzed files (LLVM finds only 526). Four rounds of tuning can further reduce sizes of individual benchmarks by up to an additional 10% and the total size is reduced by 7.05%. Also, our autotuner can reduce the size of LLVM by up to 15.21% and of SQLite by up to 10.25%.

We use all benchmarks on a AMD Ryzen Threadripper 3990X based system running Ubuntu 18.04. We based our work on LLVM version 11.0.3:\(^ {6}\) the only modifications are on InliningAdvisor\(^ {7}\).

5.2.3 Evaluation on Real-World Systems Software

We evaluate our proposed inlining autotuner for program size (Section 5) on the SPEC2017 benchmark suite; we compare against LLVM’s inlining decisions and the optimal configurations whenever they are available (Section 5.2.1 and Section 5.2.2). We also present a case study on real-world systems software: LLVM’s and SQLite’s source code (Section 5.2.3). The research questions (RQ’s) that we aim to answer are:

- RQ1: How effective is local autotuning versus LLVM’s inlining strategy on SPEC2017? (Section 5.2.1)
- RQ2: How effective is round-based autotuning? (Section 5.2.2)
- RQ3: Can local autotuning be effectively applied to real-world software? (Section 5.2.3)

Summary Results: Our autotuner can reduce the size of 14 out of the 20 SPEC2017 benchmarks by up to 27.6%; the 5 regressions can be eliminated by initializing the tuning session with LLVM’s own inlining decisions. The total reduction across all SPEC2017 files in the latter case is 4.86%; by combining the results of clean slate and LLVM-initialized autotuning the total reduction drops to 6.05%. We can find 921 (81%) optimal inlining configurations out of the 1,135 exhaustively analyzed files (LLVM finds only 526). Four rounds of tuning can further reduce sizes of individual benchmarks by up to an additional 10% and the total size is reduced by 7.05%. Also, our autotuner can reduce the size of LLVM by up to 15.21% and of SQLite by up to 10.25%.
We used the -fexperimental-new-pass-manager (it enables the new pass manager) and -Os flags. The latter is “like -O2 with extra optimizations to reduce code size.” O2 is a “moderate level of optimization which enables most optimizations.”

5.2.1 RQ1: How effective is local autotuning? We first evaluate a single autotuning session starting with a clean slate (Figure 10). Out of the 2,509 files in the SPEC2017 suite, our autotuner manages to shrink 1,306 in size, 427 remain unchanged, and 776 grow in size. The relative size of the most shrunk file is 26%, and the most inflated is 218%. The inflated files amount to a 1.39% size increase compared to the total. Out of the 20 benchmarks, 14 shrink in size with mfc having the largest improvement: 27.6%. One benchmark’s size remains unchanged, and 5 inflate; these regressions can be trivially fixed by falling back to LLVM for the inflated files. The duration of the autotuning session is 4.4 hours; a bit more than 2 hours is spent on a single file: 502.gcc/insn-attrtab.c which includes 16,178 calls. All subsequent autotuning sessions (and rounds) on SPEC2017 have almost identical runtimes.

The autotuner finds better inlining configurations for around half the files, where LLVM’s heuristic is too aggressive for binary size. In 417 others both the autotuner and LLVM find the same configurations. However, the local pair-wise scope is not enough whenever more than one call site must be considered at the same time. For example, LLVM inlines all calls in Figure 11, but the autotuner does not: inlining the individual gray/dashed edges results in size increase, however, inlining all of them triggers the callee’s Dead Code Elimination (this also eliminates its own inlined callee). Expanding the autotuner to handle these cases would be straightforward: for each callee with internal linkage and many callers, an additional configuration with all of them inlined must be checked.

We repeat the same experiment initialized with LLVM’s decisions (Figure 12); we test if these configurations are a better starting point for all of them, but it is unable to do that by examining them one-by-one. On the other hand, the autotuner can improve upon LLVM in Figure 14, a similar case to Figure 11: local inlining starting from a clean slate cannot discover dead code elimination opportunities.

Table 3: Benchmarks faring worse with LLVM-initialization.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clean slate</th>
<th>LLVM-initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>imagick</td>
<td>92.1%</td>
<td>96.3%</td>
</tr>
<tr>
<td>mfc</td>
<td>72.4%</td>
<td>79%</td>
</tr>
<tr>
<td>nab</td>
<td>97.1%</td>
<td>98.8%</td>
</tr>
<tr>
<td>nabd</td>
<td>93.9%</td>
<td>95.2%</td>
</tr>
<tr>
<td>per1bench</td>
<td>98.9%</td>
<td>99.6%</td>
</tr>
<tr>
<td>x264</td>
<td>92.3%</td>
<td>94.1%</td>
</tr>
<tr>
<td>xz</td>
<td>97.8%</td>
<td>97.9%</td>
</tr>
</tbody>
</table>

*List and descriptions of command line flags: https://clang.llvm.org/docs/CommandGuide/clang.html
Different call graphs benefit from different starting points. We can combine them by selecting the best result per call graph (Figure 15): the total reduction further improves (97.16%, 95.14%) → 93.95%, the median per benchmark (97.95%, 97.6%) → 96.4%, as well as all the other metrics.

To put the effectiveness of our autotuner in perspective, we compare it against the optimal inlining configurations (Figure 16): our autotuner finds the optimal inlining configurations in 81% of the cases, whereas LLVM only does so in 46%.

5.2.2 RQ2: How effective is round-based autotuning? Certain inlining decisions make sense only in the presence of others (e.g., Figure 11, Figure 14); we test if these can be discovered sequentially across different rounds. Each autotuning round is initialized with the resulting inlining configuration of the previous one:

1. Initial state := clean slate or LLVM’s inlining decisions.
2. Repeat n times:
   a. Autotune on top of the current initial state.
   b. Update the initial state with the previous step’s results.

We choose n = 4 as there was little gain past 4 rounds in most of our experiments. Additional rounds are clearly beneficial (Figure 17); most benchmarks improve with additional rounds, e.g., mfc 82% → 72%, leela 88.8% → 84.5%, and parest 81.8% → 77.2% (LLVM-init). Multiple rounds are necessary to discover non-local inlining configurations, i.e., those that cannot be discovered by analyzing individual call edges within one round.

An inlining configuration across rounds example is shown in Table 4: each round performs very few changes but the size decrease is significant: 100% → 71.6% → 41.2% → 41.4% → 35.8%. Despite the small size increase in round 3 (41.2% → 41.4%), the autotuner was able to reduce the size in the subsequent round by an additional 5%. This demonstrates our hypothesis that multiple rounds are an effective way of extending the autotuner’s scope. Although rarely observed in our results, this example shows that successive rounds do not always improve the results from a previous round. One solution is to select the best configuration from all the rounds.
Figure 17: Round-based autotuning versus LLVM -Os on SPEC2017. Most benchmarks benefit from additional rounds. The largest improvement is: 79% → 72% (mfc) and 112.4% → 93.4% (mfc), for LLVM-initialized and clean slate, respectively.

Table 4: 523.xalancbmk/XalanBitmap.cpp inlining changes across rounds of LLVM-initialized autotuning.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LLVM - Os</th>
<th>Round 1</th>
<th>Round 2</th>
<th>Round 3</th>
<th>Round 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>blender</td>
<td>97.6%</td>
<td>96.9%</td>
<td>96.8%</td>
<td>96.9%</td>
<td>96.5%</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>95.5%</td>
<td>95.1%</td>
<td>95.5%</td>
<td>95.0%</td>
<td>94.6%</td>
</tr>
<tr>
<td>cam4</td>
<td>96.4%</td>
<td>96.4%</td>
<td>96.4%</td>
<td>96.4%</td>
<td>96.2%</td>
</tr>
<tr>
<td>deepsjeng</td>
<td>97.6%</td>
<td>96.7%</td>
<td>96.7%</td>
<td>96.7%</td>
<td>96.5%</td>
</tr>
<tr>
<td>gcc</td>
<td>95.7%</td>
<td>95.1%</td>
<td>95.2%</td>
<td>95.2%</td>
<td>94.9%</td>
</tr>
<tr>
<td>imagick</td>
<td>96.3%</td>
<td>96.2%</td>
<td>96.2%</td>
<td>96.1%</td>
<td>96.0%</td>
</tr>
<tr>
<td>lbm</td>
<td>96.9%</td>
<td>96.8%</td>
<td>96.8%</td>
<td>96.7%</td>
<td>96.5%</td>
</tr>
<tr>
<td>namd</td>
<td>96.3%</td>
<td>96.3%</td>
<td>96.2%</td>
<td>96.2%</td>
<td>96.0%</td>
</tr>
<tr>
<td>parest</td>
<td>95.0%</td>
<td>94.9%</td>
<td>94.9%</td>
<td>94.9%</td>
<td>94.6%</td>
</tr>
<tr>
<td>perlbench</td>
<td>98.7%</td>
<td>97.9%</td>
<td>97.7%</td>
<td>97.7%</td>
<td>97.5%</td>
</tr>
<tr>
<td>pop2</td>
<td>96.0%</td>
<td>95.9%</td>
<td>95.8%</td>
<td>95.8%</td>
<td>95.6%</td>
</tr>
<tr>
<td>povray</td>
<td>99.0%</td>
<td>98.9%</td>
<td>98.8%</td>
<td>98.8%</td>
<td>98.6%</td>
</tr>
<tr>
<td>x264</td>
<td>95.6%</td>
<td>94.9%</td>
<td>94.8%</td>
<td>94.7%</td>
<td>94.5%</td>
</tr>
</tbody>
</table>

Figure 18: Round-Based autotuning clean slate (4 rounds) and LLVM-initialization (4 rounds) combined versus LLVM -Os on SPEC2017. Per benchmark relative size median: 95.65%. Per file relative size total: 92.95%.

Figure 19: Performance overhead on the (non-Fortran) SPEC-speed2017 subset: tuning inlining decisions for size (clean and LLVM-initialized combined, 4 rounds each) results in a 3.6% (geometric) mean and 2% median overhead.

Combining the 4 clean slate and 4 LLVM-initialized rounds results in an even better improvement (Figure 18): the median benchmark relative size compared to LLVM -Os is 95.65% and the per file total is 92.95%, resulting in a 7.05% improvement. Tuning for size impacts performance (Figure 19). We benchmarked the SPEC-speed2017 subset of SPEC2017 (excluding benchmarks with Fortran code as we do not tune them) and observed a 2% median and 3.6% average overhead. Interestingly, performance improved in the case of mfc, which also benefited the most from size tuning.

5.2.3 RQ3: Local autotuning applied to real-world software. We also evaluate the inlining autotuner on complex system software: LLVM’s and SQLite’s source code.

LLVM Case Study. We use the source files of LLVM’s main library components (LLVM-project/llvm/11b). The corresponding call graphs are much larger compared to SPEC2017: the median number of inlinable calls per file is 1,004 (vs 41 for SPEC2017), the maximum is 55,156 (vs 18,250), and the total number is 3,641,338 (vs 457,655).

We started the autotuning session with LLVM’s inlining configurations and ran three rounds. The total size reduction of the combined 3-round results is 15.21%; more than twice as good as the best total size reduction for SPEC2017, one reason might be that the larger and more complex LLVM-derived call graphs have more beneficial inlining opportunities. At the same time, LLVM’s complexity results in longer autotuning times: a single round takes 44-53 hours, certain files take more than 4 hours to autotune due to the large number of calls (e.g., 55,156). Nonetheless, this demonstrates that our approach is effective even on complex systems software.

SQLite Case Study. We use the SQLite Amalgamation11 for our evaluation: a single combined C file containing all the source code of the core SQLite library. It contains 18,125 inlinable calls. We evaluate two scenarios: (a) building an X86 library, (b) building a WASM library via Emscripten [27]; in both we run two 4-round autotuning sessions, one clean-slate initialized and one LLVM initialized. Each round lasted approximately 90 minutes.

- X86: The autotuned version relative sizes compared to LLVM -Os are 89.7% for clean-slate and 91.6% for LLVM-initialized. The clean-slate results are likely better because LLVM’s aggressive inlining heuristic is a bad starting point when considering size.
- WASM: The autotuned versions relative to the baseline emcc12 -Os which has inlining disabled by default are 1.26% and 0.96% smaller. Inlining as currently implemented on LLVM seems to be marginally beneficial for WASM targets; using LLVM’s own inlining heuristic result to a 18.3% size increase over no inlining, and 19.6% over the tuned version.

11https://sqlite.org/.amalgamation.html
12We used version 2.0.26 but replaced LLVM with our patched one.
We observe a substantial size reduction on X86 builds of SQLite, “The Most Widely Deployed and Used Database Engine, with likely over one trillion SQLite databases in use.” As evidenced by its own developers’ investigations, the footprint of SQLite is an important optimization target.

6 DISCUSSION
A foundation for future research on inlining: In-depth empirical understanding of optimal inlining via exhaustive evaluation is an important, practical means to guide the development of fast, effective compiler heuristics similar to how others are derived. For example, many peephole optimizations are discovered by expensive superoptimization and then incorporated into compilers [19]. Inlining autotuning can not only complement this goal, but also be used for widely deployed software, e.g., extensively tuning an important application or library (such as Chrome or SQLite) before its deployment to a large number of users/devices.

Exhaustive search for performance: Although providing the conceptual framework, our model cannot be directly used to search for optimal inlining configurations for performance. An important, general challenge is that optimality can depend on program inputs, and thus workload selection is critical. Moreover, second-order effects such as I-cache pollution can also result in implicit interactions between functions at the hardware level whenever they are executed. A possible solution is to avoid partitioning a call graph across edges that connect functions interacting in this way. It is generally infeasible to detect such interactions, but it may be possible to approximate and model them. For example, profiling can highlight frequent calls that should be marked as “never partition”. Studying the impact of such effects and the need to model them, as well as how to balance between performance and code size, is an interesting research direction.

Learning inlining heuristics: Our work provides the foundation for generating large amounts of data via scalable exhaustive search to enable developing effective ML models for inlining. Prior work has considered learning a heuristic for program size [5, 9, 17, 18, 26]; the training data was generated by various exploration methods. However, none considered (or had access to) the optimal decisions. Good training data is necessary and critical to enable such research.

Autotuning scalability: Although the evaluation of our proof-of-concept autotuner demonstrates its usefulness, scalability was not our primary goal. A practical implementation can take advantage of multiple properties to reduce the number of necessary evaluations, and as a result the tuning time; e.g. only re-tuning parts of call graphs that change between rounds, or by taking advantage of the independence properties described in section Section 3.2 to combine multiple rounds into one.

7 RELATED WORK
This section surveys related work, which we categorize into several threads: (1) empirical studies on inlining, (2) mitigating the complexity of inlining, (3) inlining heuristics, (4) search space exploration, (5) machine learning heuristics, and (6) outlining.

Empirical studies on inlining: Several efforts exist that aim to empirically investigate different aspects of inlining, including comparisons of static and profile-based heuristics [3, 16, 25], studying a particular inliner and its effects on programs [10, 12], and evaluating inliners under specific contexts, such as ARM-based embedded systems [2]. These studies focus on comparing existing inlining heuristics. In contrast, we aim to study optimal inlining by understanding the inlining search space and deriving a roofline analysis.

Mitigating the complexity of inlining: One of the difficulties with making good inlining choices is predicting the cascading effects of the additionally enabled transformations. Even the order at which call sites are considered for inlining can have a significant impact on the resulting code [7]. Inlining trials [11] attempts to sidestep this issue by tentatively inlining functions to estimate the impact of subsequent optimizations more easily. An alternative approach for VM-based languages is to propagate arguments and their types across function calls [24]; the aim is to reduce the complexity of predicting which choices lead to further optimizations. At a high level, our (round-based) autotuner operates under a similar principle of “trials”. However, it is not meant to run as part of a regular compilation, thus, it is not constrained by a strict time budget and it can explore a much larger number of candidates.

Inlining heuristics: Finding effective heuristics for inlining has been the subject of research for decades. Static heuristics use source-code-derived information [15, 28]. Various profile-guided/JIT-based heuristics that take advantage of runtime information exist [8]. Partial inlining of a method’s hot path simplifies the complexity of inlining [1]. Using additional context information, such that a certain virtual call is mostly made with one or two concrete types, can facilitate better inlining choices [14]. Profile information can be used to better estimate the trade-offs between performance and increased program size [28] or increased compilation time [4]. Combinations of profiling information, clustered inlining, and trials have also been proposed [21]. Unlike these, our work focuses on deriving an inlining autotuner that evaluates 100-1,000s of different inlining choices, instead of one or several candidates.

Inlining search space exploration: Previous attempts at search space exploration focus on different aspects of inlining. One approach is to tune a heuristic’s parameters (e.g., the inlined callee’s number of statements threshold) via genetic algorithms [5]. This expands (or focuses) the space of potential inlining configurations selected by the heuristic. Adaptive inlining explores the space of potential heuristics [9]: the space is defined by a set of program metrics (e.g., statement count and constant parameter count) and rules on them (e.g., calls in loops whose callees have fewer than x statements should be inlined); the rules are tuned via hill-climbing. Both approaches focus on performance and do not aim to be exhaustive. Our work targets optimal inlining and focuses on a different kind of exploration: the space of all potential inlining configurations and their impact on program size.

---

https://sqlite.org/mostdeployed.html
https://sqlite.org/footprint.html
Machine learning inlining: Machine learning has been suggested as an alternative to “hand-crafted” inlining heuristics. Techniques such as random forests [18] and NeuroEvolution of Augmenting Topologies (NEAT) [17] have been used for VM-based languages. Up until now, such techniques have not been widely used in production compilers such as LLVM and GCC. However, a recent reinforcement learning-based approach, MLGO [26], can be optionally enabled in LLVM.

Outlining: Outlining is the opposite of inlining, a part (sequence of instructions) of a function is replaced with a call to a newly formed function. Chabbi et al. [6] introduce a round-based outline for code size reduction. The proposed outline operates at the ISA-level, after IR-level optimizations (including inlining), and could be used in combination with our autotuner to further reduce code size.

8 CONCLUSION
We have presented an extensive, empirical investigation into optimal function inlining on the SPEC2017 benchmark suite. To make our study feasible, we have introduced a novel inlining search space formulation that allows massive space reductions (from 2^249 to 2^25 on SPEC2017). Our optimal inlining analysis on more than 1,000 SPEC2017 C/C++ files shed light on an inlining roofline and quantified the opportunities for improving state-of-the-art inlining strategies. Our study has also led to actionable insight, which allowed us to design a simple yet effective autotuner that outperforms LLVM’s inlining heuristic not only on SPEC2017 benchmarks (by up to 28%) but also on LLVM’s codebase (by 15%) and on SQLite (by 10%). Our autotuner is embarrassingly parallel and can be used in “compilation farms” to rapidly reduce the program size of relevant applications. We expect our results and methodology to help further understand the inlining search space and develop better heuristics for program size as well as performance.

REFERENCES
[24] Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in line, please! exploiting the indirect benefits of inlining by accurately predicting further inlining. In Proceedings of the compilation of the co-located workshops on DSM@I, TMC@II, AGERE! 2011, AOOPE's'11, NEAT@I, & YML@I. 317–328.

A ARTIFACT APPENDIX

A.1 Abstract
The artifact contains the code and dataset we used for our experiments, as well as scripts to generate the numbers, figures, and tables of our evaluation. Specifically, it includes (a) the LLVM-IR files we used both for exhaustive search and autotuning (b) a modified LLVM that we use for exhaustive search and autotuning; (c) scripts to run exhaustive search and autotuning; (d) the expected outputs; (e) scripts to generate the tables and figures of our paper; (f) scripts to perform exhaustive search and autotuning only on smaller call
graphs and to validate the results against the provided ones. Everything is packaged and pre-built as a docker image. A standard X86 Linux machine running docker is necessary to evaluate this artifact.

### A.2 Artifact check-list (meta-information)
- **Data set**: LLVM-IR derived from SPEC 2017 CPU benchmarks
- **Run-time environment**: Linux
- **Hardware**: X86 computer
- **Output**: Autotuning results, exhaustive search results, figures and tables.
- **Experiments**: Exhaustive search on a subset of SPEC 2017 CPU benchmarks, autotuning on all of them.
- **How much disk space required (approximately)?**: 30G
- **How much time is needed to prepare workflow (approximately)?**: A few minutes to download and import the docker image.
- **How much time is needed to complete experiments (approximately)?**: Several days to fully reproduce the results (even on a modern 64-core machine), several tens of minutes to validate the provided results.
- **Publicly available?**: Yes
- **Code licenses (if publicly available)?**: MIT
- **Archived (DOI)**: 10.5281/zenodo.5848986

### A.3 Description

#### A.3.1 How to access.
The artifact can be downloaded from [https://doi.org/10.5281/zenodo.5848986](https://doi.org/10.5281/zenodo.5848986)

#### A.3.2 Hardware dependencies.
A standard X86 computer. Fully reproducing the exhaustive search results requires significant amounts of main memory, around 16GB per parallel job, due to the call graph size of certain auto-generated files in SPEC2017.

#### A.3.3 Software dependencies.
Docker.

#### A.3.4 Data sets.
Included in the docker image.

### A.4 Installation

tar xf ASPLOS22-Inlining-Artifact.tar.gz

cat inlining-artifact-image.tar | docker import - inlining_artifact

### A.5 Evaluation and expected results

The exhaustive search results (Section 4). The autotuning results (Section 5.2). The paper figures related to exhaustive search and autotuning. The instructions are in README.md (included both in the root directory of the docker image and in the .tar.gz file).