Inlining for Code Size Reduction

Vinícius S. Pacheco[1]
PUC-MG
Brazil
vinicius.pacheco@sga.pucminas.br

Thaís R. Damásio[1]
PUC-MG
Brazil
thais.damasio@sga.pucminas.br

Luís F. W. Góes
University of Leicester
UK
fabricio.goes@leicester.ac.uk

Fernando M. Quintão Pereira
UFMG
Brazil
fernando@dcc.ufmg.br

Rodrigo C. O. Rocha
University of Edinburgh
UK
rrocha@ed.ac.uk

Abstract

Function inlining is a compiler optimization that replaces the call of a function with its body. Inlining is typically seen as an optimization that improves performance at the expenses of increasing code size. This paper goes against this intuition, and shows that inlining can be employed, in specific situations, as a way to reduce code size. Towards this end, we bring forward two results. First, we gauge the benefits of a trivial heuristic for code-size reduction: the inlining of functions that are invoked at only one call site in the program, followed by the elimination of the original callee. Second, we present and evaluate an analysis that identifies call sites where inlining enables context-sensitive optimizations that reduce code. We have implemented all these techniques in the LLVM compilation infrastructure. When applied onto MiBench, our inlining heuristics yield an average code size reduction of 2.96%, reaching 11% in the best case, over clang -Os. Moreover, our techniques preserve the performance gains of LLVM's standard inlining decisions on MiBench: there is no statistically significant difference in the running time of code produced by these different approaches.

Keywords: Code Size, Function Inlining, LTO.

1 Introduction

Compilers optimize programs along different dimensions of efficiency, examples of which include running time, power consumption and code size. The last one of them is the focus of this paper. Reducing code size is important in at least three scenarios. Embedded [6] and mobile [3] systems being the most well-known. Yet, code compression can also be of capital importance in servers. As an example, the large binaries produced as the result of translating PHP to C++ was one of the reasons behind the appearance of HHVM in the Facebook software stack [13]. Furthermore, reducing code also tends to speedup programs, as it maximizes the amount of hot code that fits into the instruction cache [4, 14].

Function inlining is typically seen as an optimization that improves running time at the cost of increasing code size. This intuition is true in general: by replacing calls with a copy of their function body, code tends to increase [5]. This growth can be as dramatic as 100-fold, as recently observed by Poesia and Pereira [15] on SPEC CPU2017's `deepsjeng`. Therefore, much of the effort that goes into the craft of state-of-the-art inlining heuristics is aimed at maximizing speed while preventing code-size explosion [2, 12, 23, 24].

Breaking with Folk Intuition. This paper departs from what we call "folk intuition": the well-established idea that inlining trades performance for size. While we acknowledge that, in general, such is the case, we demonstrate that, when properly engineered, inlining can be used specifically to reduce code size. This paper starts by analyzing a straightforward heuristic: inlining—and eliminating—functions that are called only once in the whole program [1]. This optimization can be employed more aggressively at link time. As we show in Section 4, this inlining heuristic is effective at reducing code size. Afterwards, we extend this inlining heuristic a step further. We perform an *a posteriori* analysis of inlined code (instead of the traditional *a priori* approach). In other words, we analyze the cost of inlining based on the effects of inlining on the code after it happens, and not before, as typically performed.

Summary of Contributions. This paper is the first to promote inlining for code-size reduction. Inlining for code-size reduction is different of limiting the amount of code bloat while inlining for performance. The proposal of using inlining to reduce code size is counterintuitive, because inlining a function duplicates its body; hence, suggesting code expansion. To achieve compression, we introduce an optimistic

[1]The first two authors contributed equally to this work.

SBLP'21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9062-0/21/09…$15.00
https://doi.org/10.1145/3475061.3475081

strategy that first inlines a function call into a cloned version of the caller. This heuristic then analyzes if inlining enables further optimizations that would reduce code. If that is not the case, then it backtracks to the original function without inlining. This enables the compiler to use one level of context-sensitiveness to take inlining decisions. Similar to prior work [20], we are taking future optimizations into account, in an explicit manner, to decide the profitability of a given transformation. We also evaluate, in the context of code size reduction, the well-known strategy of inlining functions with a single call site, as previously mentioned. In Section 4, we show that the combined solution reduces MiBench's [9] binaries by up to 11% (2.96% on average), when compared to clang -Os.

2 Motivation

Inlining must be applied carefully. Excessive inlining, on the limit, might augment the program by an exponential factor [15]. This code explosion compromises program execution, as it leads to poor locality in the instruction cache. Therefore, the main challenge for inlining heuristics consists in determining which function calls should be inlined.

2.1 When Inlining Reduces Code Size

Inlining will replace the body of a function at its calling sites. In face of no optimization, program size will increase if the function body contains more instructions than what is necessary to implement the call. These instructions—that implement the function call—copy *actual* into *formal* parameters, and invoke the function itself, e.g., through a `call` operation. Naturally, if they outsize the function body itself, inlining leads to code-size reduction. The next example illustrates this possibility.

Example 2.1. Listing 1 shows a code snippet from MiBench's `lame`. Function *lame_init_infile* calls *GetSndSamples*. When compiled to x86, the binary representation of this call outsizes the body of *GetSndSamples*. Thus, inlining the call to *GetSndSamples* leads to trivial code reduction.

```
1  static unsigned long num_samples;
2
3  unsigned long GetSndSamples(void) {
4        return num_samples;
5  }
6
7  void lame_init_infile(lame_global_flags *
       gfp) {
8    count_samples_carefully=0;
9    OpenSndFile(gfp,gfp->inPath,gfp->
       in_samplerate,gfp->num_channels);
10   if (GetSndSampleRate()) gfp->
       in_samplerate=GetSndSampleRate();
11   if (GetSndChannels()) gfp->num_channels=
       GetSndChannels();
12   gfp->num_samples = GetSndSamples();
13 }
```

Listing 1. Snippet extracted from file *get_audio* in `lame` benchmark.

Inlining gives the compiler the chance to run context sensitive optimizations. These optimizations might reduce code size. Therefore, in face of optimizations, it is possible that, even when the function body outsizes the code to invoke the function, the binary still decreases. However, identifying these scenarios is a non-trivial task, as Example 2.2 shows.

Example 2.2. Assume that function *f1* calls function *f2* in Listing 2 with a null pointer as first argument. In this case, *f2* returns zero, regardless of other inputs. Under such circumstances, a combination of inlining, constant propagation and dead-code elimination eliminates the call to *f2* at Line 10 of Listing 2. When applied on x86, this inlining reduces the code of *f1* from 34 instructions (83 bytes) down to 11 (25 bytes). Even more reduction is possible, if the compiler can prove that *f2* is not used elsewhere. In this case, this function could be removed from the executable program.

```
1  int f2(int *V, int n, int factor) {
2      int valid = V?1:0;
3      int s = 0;
4      for (int i = 0; i< (n*valid); i++)
5          s += factor*V[i];
6      return s;
7  }
8  int f1(int* W, int *V, int n, int t) {
9      for (int i = 0; i<n; i++) {
10         W[i] = t*i + f2(0,n,t);
11     }
12 }
```

Listing 2. Example of a function where the current version of inline does not detect it as profitable.

The current inlining heuristic used in LLVM fails to identify opportunities like those seen in Example 2.2. It prevents inlining *f2* due to a threshold that limits code bloat. The inliner measures the size of the control-flow graph from function *f2* prior to constant propagation and dead-code elimination, missing the fact that the inlined code would be fully optimized away. The goal of this paper is to incorporate some level of code simplification in the inlining analysis, using the calling context to decide whether inlining is profitable. The concretization of this idea is the subject of the next section.

3 Inlining for Code-Size Reduction

As already discussed in Section 2, there are situations in which inlining leads to code-size reduction. This paper benefits from two of them, which we describe below and explain in the rest of this section:

- **Solitary**: the callee is invoked at one program site.
- **Situational**: context sensitive optimizations trim the caller-callee pair.

3.1 Inlining of Solitary Calls

A *solitary call* is the invocation of a function that has a single call site in the program. If this function is inlined, then code size reduction is expected, as long as the compiler is allowed to remove the original definition of the function. In this case, only one instance of its body will remain in the optimized program. This is known as dead function elimination [1]. Such elimination is possible under two circumstances:

1. The function is private to the current compilation unit, e.g., marked as *static* in C. Private functions cannot be referenced from an external translation unit. Once inlined, the function will be considered an *unused global symbol*, and can safely be removed from the object file.
2. Inlining happens at *link time*. At that moment, the compiler has access to the whole program, therefore all functions are guaranteed to have no external use, if not compiling a library. Hence, functions with a single call site can be eliminated after inlining[1].

Inlining solitary calls is a well-known heuristic, used in several programming languages [1]. LLVM implements this heuristic in its inlining analysis as a bonus constant that is discounted from the cost of inlining a given call site[2]. In this paper, we analyze this heuristic in isolation, as well as combined with our *situational* heuristic, as a key strategy when inlining for code size reduction.

Inlining and Register Pressure. Although code-size reduction is the expected effect in any of the two cases above, it is not certain. In our experiments, we have observed that code could still increase, even when the original copy of the inlined function was eliminated. One situation that provokes this counterintuitive behavior is *register allocation*. Inlining of large functions may increase register pressure, because more variables might be simultaneously alive at some program point. Due to the extra register pressure, more load and store instructions will be inserted into the architecture-dependent version of the program. The `adpcm(c)` benchmark is one such example (see Figure 3).

Register pressure bears no impact on the size of the LLVM intermediate representation: register allocation happens once this representation is lowered into machine code [17]. Thus, inlining of solitary calls never increases the size of the program's intermediate representation, but can increase the size of the final executable. To mitigate this effect, we limit the size of the callee function to 750 LLVM instructions. This threshold is arbitrary, and is based on the observation that large functions are more likely to introduce a significant register pressure after inlining. In future work, a more detailed analysis can be introduced to avoid inlining functions that may cause a significant increase in register pressure.

3.2 Optimistic Inlining with Rollback

Inlining enables *context-sensitive optimizations*. The *context* of a function call is determined by the sequence of functions active when that invocation happens. Context-sensitive optimizations can be very effective. Quoting Poesia and Pereira [15], "We have found speedup opportunities with our naïve context-sensitive constant propagation algorithm. For instance, in the `Dhrystone` benchmark, we could produce a binary 32x faster than `clang -O2`. In this case, the combination of constant propagation, loop unrolling and dead code elimination let the compiler solve much of the computations in the benchmark statically." Example 2.2, earlier seen in Section 2 already illustrates the benefits of context-information for code-size reduction. Nevertheless, in spite of these benefits, compilers do not use context-sensitive information when determining the profitability of function inlining. This omission is due to the fact that obtaining such information is a costly endeavour.

The inlining heuristics from mainstream compilers compute costs *a priori*, i.e., before inlining effectively takes place. Yet, the product of a context-sensitive optimization can only be observed *a posteriori*, i.e., after inlining happened. In this section, we propose a strategy based on rollback that changes this perspective; therefore, effectively implementing an inlining heuristic that considers information a posteriori. The rollback strategy exploits the static aspect of code-size optimizations. Figure 1a summarizes the flow of typical inline heuristics and Figure 1b highlights the clone and rollback steps that we propose in this paper.

A Posteriori Cost Evaluation. When the standard inlining analysis calculates the costs to decide whether or not it will be profitable to inline a function, the compiler does not have much information about the context of the caller-callee pair and therefore cannot identify opportunities for future optimizations. As shown in section 2.1, it is insufficient to analyse the two functions separately. In some cases, it is only visible that inlining is profitable after we have already inlined and applied further optimizations.

Our solution always inlines, rolling back when inlining is deemed unprofitable. For a given call site, we create a clone of the caller function with the callee inlined. Then, we apply standard compiler optimizations to simplify the inlined code

[1] When compiling a library, dead function elimination should be restricted to circumstance (1). In this case, non-static functions might be called by external users. Thus, we cannot remove them even at *link time*.

[2] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Analysis/InlineCost.cpp#L1590

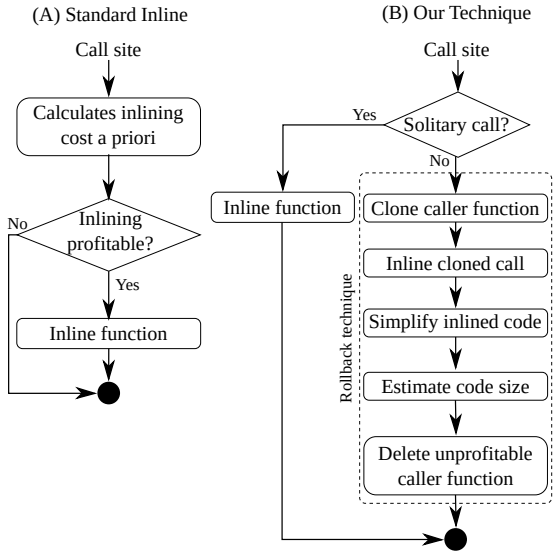

Figure 1. (A) LLVM's standard heuristic. (B) Our approach that combines the solitary and rollback heuristics.

in the context of the cloned function. These optimizations include: constant propagation, code simplification, dead-code elimination, and loop elimination. Once we have both versions of the caller function — original and optimized — we use an *objective function* to decide which one to keep.

The objective function estimates the code-size benefit of replacing the original by its optimized counterpart. In order to estimate the code-size benefit, we first compute the code-size cost for all instructions in each version of the function. Therefore, inlining is profitable if the code-size cost of the original function is greater than that of the optimized one. If inlining is deemed unprofitable, we keep the original function, otherwise, we keep its optimized version.

However, one LLVM's IR instruction does not necessarily translate to one machine instruction. Thus, the profitability is measured with the help of the compiler's target-specific cost model. The actual cost of each instruction comes from querying this compiler's built-in cost model, which provides a target-dependent cost estimation that approximates the code-size cost of an IR instruction when lowered to machine instructions. Our implementation makes use of the code-size costs provided by LLVM's target-transformation interface (TTI). This cost model is used by several optimizations implemented in LLVM [16, 18, 19, 22].

The impact of the rollback strategy on the compilation time depends on two key factors: First, the number of call sites that must be analyzed by the rollback analysis. The larger the number of call sites, the larger the number of clones and simplifications that must be performed, therefore increasing compilation time. Second, the size of the caller and callee functions, since larger functions will take longer to be both cloned and simplified. Therefore, in order to limit the compilation time overhead introduced by this analysis, we use a threshold on the size of the caller-callee pair. A good threshold value will limit both factors by filtering out all call sites involving caller-callee pairs of large functions. In our evaluation (see Section 4), we show this threshold can significantly reduce the impact on compilation time, with a negligible impact on our code size reduction.

3.3 Link-Time Optimization

There are different ways of applying the proposed inlining heuristics, with different trade-offs. For instance, inlining can be applied per compilation-unit. This modus operandi leads to lower compilation-time overheads as only a small part of the program is considered at each moment. However, inter-procedural optimizations, such as inlining, are also limited to operate within a single translation unit at a time. Thus, it prevents inlining when the caller-callee pair is defined in different source files. Moreover, the compiler must also be more conservative when removing callee functions with no use after inlining, since these functions may still be used externally, unless they are explicitly defined as internal to its translation unit.

In contrast, link time optimizations (LTO) are performed after all translation units have been linked into a single monolithic module, as shown in Figure 2. As a result, when performed at link-time, inlining can be applied on the whole program, exposing more optimization opportunities [11, 21]. Moreover, by having access to the whole program at once, the compiler is also able to internalize global definitions that were originally defined as externally available. Thus, enabling the inliner to be more aggressive when removing callee functions that become unused after inlining.

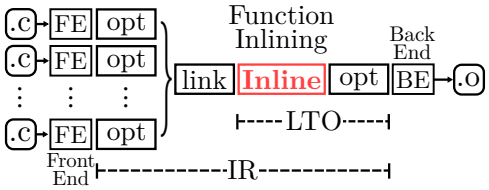

Figure 2. The LTO-based compilation pipeline.

Figure 2 shows an overview of the compilation pipeline used throughout our evaluation, presented in Section 4. First, we have the front-end and early code-size optimizations applied to each compilation unit, where we use `clang` with a default optimization level (e.g., `-Os`), but no inlining is performed. Then, inlining is applied, in full LTO mode [11], after all translation units are linked into a single module. Finally, some late code-size optimizations and the back-end are applied to the optimized code, again using `clang` with an optimization level and inlining disabled.

4 Evaluation

This section evaluates the effectiveness of the inlining heuristics discussed in Section 3. This evaluation shall be guided by the following research questions:

- **RQ-A**: What is the impact of our inlining heuristics on the size of the binaries that LLVM produces?
- **RQ-B**: How do our inlining decisions differ from those taken by the standard LLVM inliner?
- **RQ-C**: What is the impact of our inlining heuristics on the time that LLVM takes to compile programs?
- **RQ-D**: What is the effect of our inlining heuristics on the execution time of programs that LLVM generates?

4.1 Experimental Setup

Software. Results reported in this section are produced on Linux 20.04, released on April 23rd, 2020. The heuristics presented in Section 3 are implemented in LLVM v11. This version of LLVM is also used as the experimental baseline.
Hardware. Experiments run on an Intel i7-8750H 2.20-4.10 GHz CPU (Central Processing Unit), 1TB HDD with 5400 RPM, and 32GB 2666MHz of RAM.
Benchmarks We use MiBench embedded benchmark suite [9] to perform our experiments. This choice is motivated by the goals of this work: since its conception, MiBench has been a staple in discussions about code-compression.
Contending Approaches. In this section, we compare five different approaches to generate code:

- **Baseline**: clang using the code size optimization (`-Os`) with inlining disabled.
- **Standard**: the same as *baseline* but with inlining enabled, using LLVM's inlining heuristics.
- **Solitary**: the baseline augmented with the inlining heuristic for solitary calls (from section 3.1).
- **Rollback**: the baseline augmented with the rollback strategy (proposed in section 3.2).
- **Combined**: the baseline with both the solitary and the rollback heuristics combined.

Methodology Results for program speed and compilation time are computed based on the execution of 51 samples. We exclude the first execution of each benchmark per approach (the warm-up run). We also remove the slowest and the fastest executions. Thus, results are the *arithmetic* average of 48 samples. We adopt a confidence interval of 99%. Code size is measured as the size of the text segment of the binary file produced at the end of the compilation pipeline (after linking). Code size does not vary across samples; hence, the confidence level is immaterial in this case. Charts that report relative results use clang `-Os` without inlining as the baseline. To prevent inlining, we created a new flag that completely disables the Inliner pass. We use it instead of `-fno-inline-functions`, because the latter still permits inlining. Our new flag is passed to clang in the early and late code-size optimizations, as explained in section 3.3.

4.2 RQ-A: Code Size Reduction

Figure 3 compares the size of binaries produced by the different inlining approaches. While the existing inlining heuristic in LLVM corroborates the "folk intuition" that inlining increases code size, as it causes an average increase of over 1% in the program binary, our inlining heuristics designed specifically for code compression are able to achieve an average reduction of almost 3%. We conclude that *this result confirms our thesis that a properly engineered inlining heuristic can be effective for code size reduction.*

Inlining solitary calls already leads to code reduction, achieving up to 7.2% of compression on `rsynth` and 2.51%, on average, across all benchmarks. This heuristic yields code growth only in `adpcm(c)`, where an increase in register pressure leads to more stack manipulation in the binary of this benchmark. Notice that this expansion happens only at machine-code level; not in the LLVM intermediate-representation level. Our rollback heuristic brings less benefit when used alone: on average, we observe reductions of 1.18%. However, in some benchmarks, such as `basicmath`, results are very significant: rollback attains a reduction of 11%. Interestingly, the inlining of solitary calls bears no effect onto this particular benchmark. Therefore, no heuristic strictly dominates the other. Nevertheless, the best result is achieved by combining both heuristics, producing an average reduction of 2.96%. In other words, once combined, the heuristics generate code that is, on average, 2.96% smaller.

4.3 RQ-B: Qualitative Analysis

Figure 4 compares the inlining decisions performed by LLVM with the decisions performed by the heuristics from Section 3. Clearly, LLVM's standard inlining heuristic leads to substantially more inlining than our approach does. In absolute numbers, the standard heuristic inlined 7,307 function calls, whereas we inlined 1,748. Figure 4 also details the amount of inlined calls that are common for both the standard and the combined heuristics, as well as those that are unique to each heuristic. Overall, our techniques have decided to apply inlining at 1,264 common call sites. We have also inlined 484 calls that LLVM has forgone. Therefore, *we conclude that our combined heuristic is not performing less inlining by simply restricting the standard heuristic.* Rather, it achieves code-size compression due to its improved decision-making analysis.
On the Size of Inlined Functions. The number of analyzed call sites may vary depending on the different thresholds used by each heuristic as well as their inlining decisions. These thresholds limit the size of callees when performing inlining (as explained in Section 3.1). Surprisingly, we have noticed that the sizes of the inlined callee functions tend to be larger for our heuristics focused on code-size reduction, which seems counter-intuitive. Nevertheless, besides the difference in their thresholds, there are other fundamental differences among the inlining techniques that can also

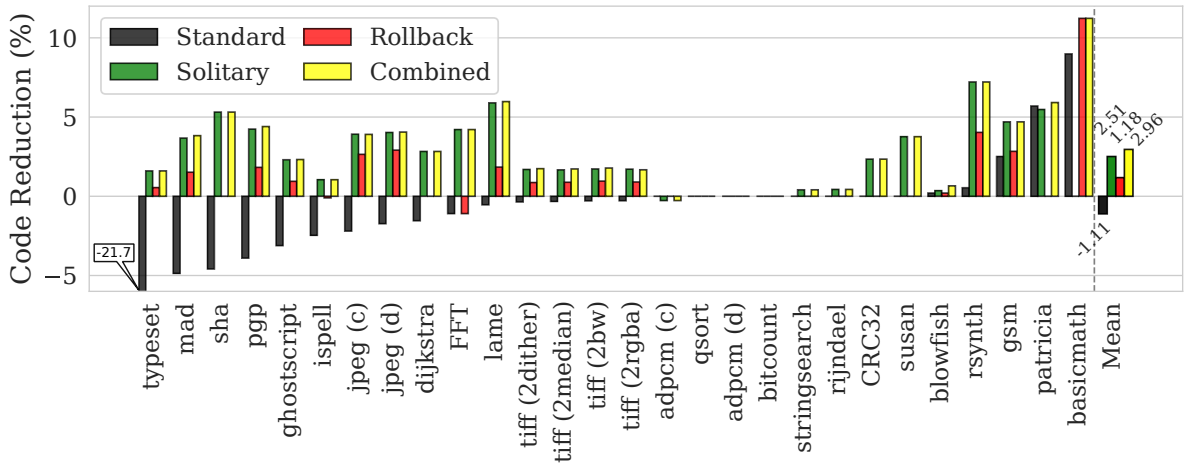

Figure 3. Variation of code size produced by the different inlining approaches. Results are relative to LLVM `-Os` with no inlining in LTO mode. Positive bars denote percentage of size reduction; negative bars, percentage of size expansion. Averages using `-Oz` as the baseline are similar, with Standard=-0.57%, Solitary=+2.66%, Rollback=1.18% and Combined=+3.08%.

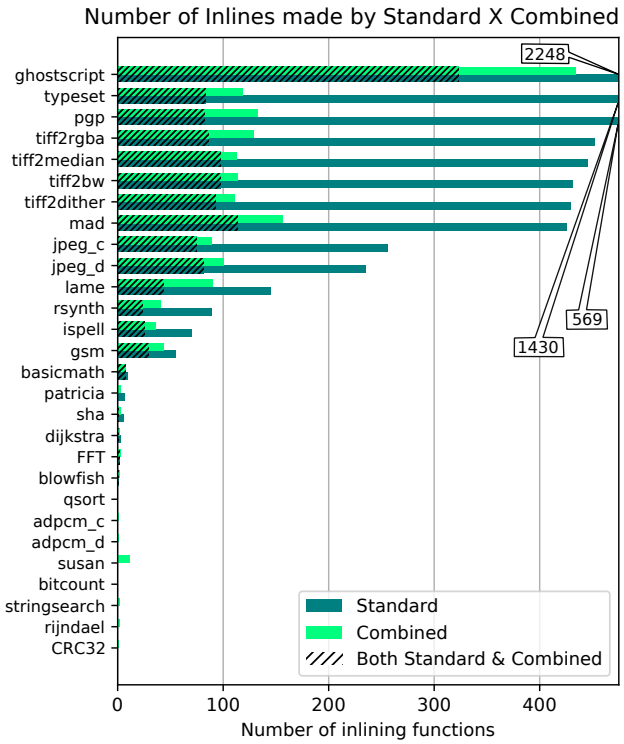

Figure 4. Number of functions inlined by the LLVM's Standard inlining heuristics and our Combined (Solitary plus Rollback) approaches.

impact on the size of the callee function. The inlining of solitary calls happens if: (i) the callee is invoked at only one site; and (ii) the callee can be removed if unused. This heuristic is limited by the code size of the callee, based on the adopted threshold; however, the threshold is large. As a result, it ends up choosing callee functions with the largest average sizes. However, because only one copy of the function is kept, it is still very effective at reducing code size.

Unlike the inlining of solitary calls, the rollback technique directly estimates the size of the caller function after inlining to analyze its profitability. However, as shown in Section 2.1, inlining can enable several optimizations capable of simplifying the inlined code, possibly even reducing complex control-flow graphs to a single instruction. Therefore, the size of the original callee function does not translate directly to the size of the inlined code. This observation can be confirmed empirically by inspecting Figure 3.

The Effect of Inlining on Large Programs. Poesia and Pereira have reported that the larger the program, the higher the percentage of code-growth caused by inlining [15]. Our experiments corroborate this observation. Even though the MiBench programs are relatively small, their sizes vary considerably. On average (arithmetic mean), binaries have 155.75 KB. One of the largest programs in our collection is `typeset`, whose binary contains 572.44 KB distributed across 452 functions. LLVM's standard heuristic performed inlining at 1,430 call sites, leading to a code expansion of 21.7%. Similar effects have been observed in `ghostscript`, the largest program with 1,171.31 KB, as well as the other large benchmarks such as `mad` and `pgp`. In contrast, three different heuristics that stem from this work—inlining of solitary calls, inlining with rollback and their combination—led to code reduction in these two benchmarks (see Fig. 3).

4.4 RQ-C: Compilation Time

Figure 5 shows how compilation time varies depending on the inlining heuristic adopted. On average, the time taken by the inlining of solitary calls is statistically equivalent to the

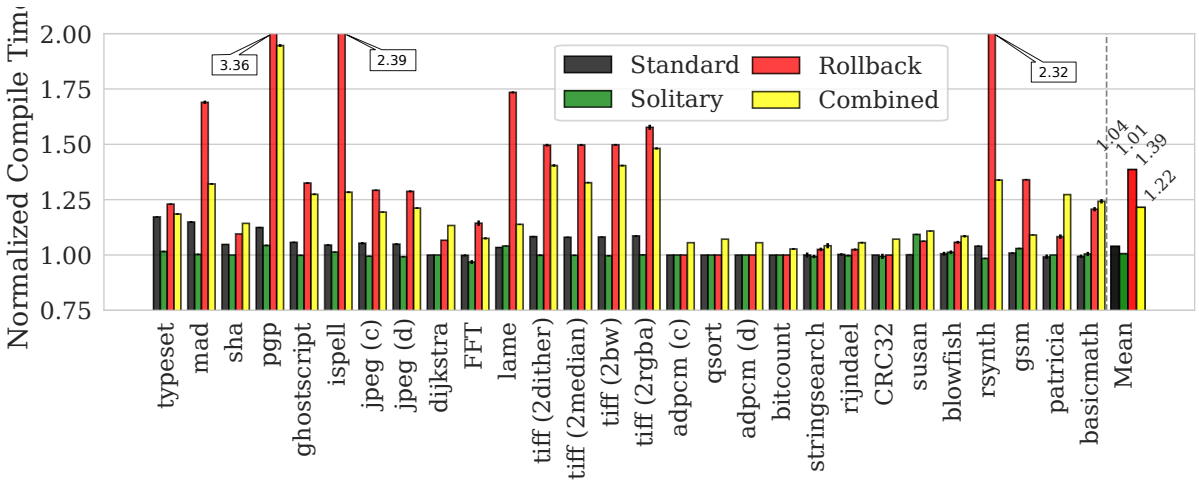

Figure 5. Compilation time of different inlining approaches. Results are relative to LLVM `-Os` with no inlining in LTO mode. Positive bars show growth in compilation time (in number of times).

baseline with no inlining, given a confidence level of 0.99. The rollback heuristic, on the other hand, has an average compilation overhead of 39%. The biggest slowdowns come from pgp and `ispell`, which we shall discuss later.

The Impact of a Cut-Off Threshold. Without the rollback threshold, the application of our heuristic onto `typeset` and pgp would slowdown compilation by over 10x. These are the two programs with the largest number of call sites. However, our rollback threshold of 750 LLVM instructions is enough to significantly reduce the compilation overhead. This threshold skips 45% of the call sites involving the largest pairs of caller-callee functions in MiBench. Figure 6 shows the call sites that have been skipped. Even though pgp has the second largest number of call sites, most of them involve functions below the cut-off; hence, only 10% of its most expensive call sites are skipped. The `ispell`, `rsynth`, and `lame` benchmarks follow a similar pattern, as shown in Figure 6. Only a tiny percentage of their call sites are cut off by the threshold. Moreover, a significant percentage of the call sites that go through the rollback heuristic have a combined size over 500 LLVM instructions; hence, being close to the threshold. This pattern contributes to a larger overhead when compared to other benchmarks. Interestingly, once these two heuristics are combined, the total compilation time is less than when rollback is used alone. The explanation for this behavior is that the functions removed by our first heuristic, the inlining of solitary calls, reduces the amount of code that must be inspected by the rollback heuristic.

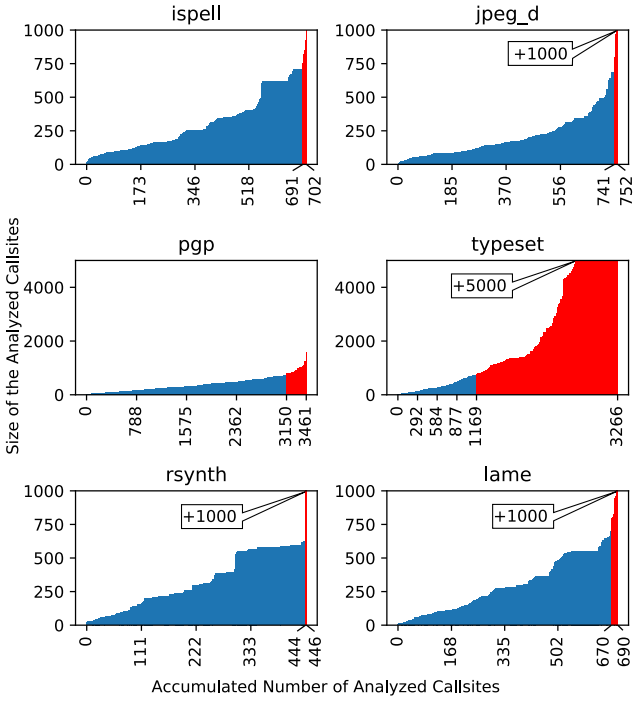

Figure 6. Call sites sorted by the size of their caller-callee functions. The red bars show the call sites that are skipped based on the rollback threshold.

4.5 RQ-D: Execution Time

When restricted to the MiBench suite, our evaluation of execution time shows that inlining has little impact on performance. On average, the standard inlining heuristic has a speedup close to 1.01 over the baseline LLVM `-Os` without inlining. This is somewhat expected since the longest running programs execute for only about half a second.

Our combined inlining heuristics have a negligible impact on the average speedup. Although the standard inlining heuristic adopted by LLVM yielded slightly better results than the other approaches considered in this paper, this difference cannot be considered statistically significant within a

confidence interval of 99%. Applying the Student Test on the runtime measurements from MiBench, then we have that the p-value between the *standard* heuristic and the baseline is 0.14. The p-value between the *solitary* heuristic and the baseline is 0.62. The p-value between the *rollback* heuristic and the baseline is 0.65, and the p-value between the *combined* heuristic and the baseline is 0.51. All these values are above the confidence level of 0.01; hence, they cannot be considered statistically different.

Nevertheless, exceptions to these results could be observed in individual benchmarks. In particular, inlining with rollback (Section 3.2) greatly improved the speed of one of the versions of `tiff` (the so called `2median` version). Similarly, the inlining of solitary calls yielded statistically significant speedups in `ispell` and `ghostscript` (our largest benchmark). However, it also led to regressions in `blowfish`. These results are consequence of correct and wrong inlining decisions. Given that our objective function was tuned for code-size reduction, this behavior in performance is to be expected. From this discussion, we conclude that, at least when restricted to the MiBench programs, *our heuristics can reduce code-size without leading to performance degradation.*

5 Related Work

Optimizations for Code-Size Reduction. Code-size reduction has never been a top concern in the implementation of mainstream compilers—program speed, instead, appearing as a core driving force. Nevertheless, the year of 2020 has seen much interest in the craft of optimizations for code compression, at least in the LLVM community. Evidence to this fact appears in recent industrial and academic work [3, 7, 8, 10, 18, 19] and in the notes of the 2020 LLVM developers meeting. For instance, quoting from the minutes, "`opt -Oz` *does not disable neither loop unrolling nor inlining*", which are transformations that usually cause code growth. Interestingly enough, inlining emerges as a culprit for code growth—a fact that is not always true, as this paper demonstrates.

Customization of Inlining Heuristics. Descriptions of inlining heuristics abound. Incidentally, the vast majority of them use code-size as a negative cost when determining when inlining is profitable. For an overview on this approach, we recommend the work of Leupers and Marwedel [12] and Zhao and Amaral [24]. The discussion carried out by Xinrong et al. [23] summarizes well the perception of the compiler community with regards to inlining: "*Inlining functions can eliminate the overhead which is resulted from function calls, but with inlining, the code size also grows unpredictably; this is not suitable for embedded processors whose memory size is relatively small.*" We emphasize that such a stance is not the position adopted in this paper. Even though we are aware that unrestricted inlining may lead to code-size explosion, we have demonstrated that, when applied carefully, inlining can be a means to reduce code.

Inlining in Mainstream Compilers. Implementations of inlining heuristics consider code size *a priori*, instead of *a posteriori*. That means that when feeding an objective function with code-size information, researchers count the function body N times, where N is the number of times that said function can be inlined. As an example, in LLVM, function arguments subtract one unit from the inlining cost, because it is assumed that one instruction is necessary to copy an actual into a formal parameter. Each instruction in the would-be inlined function adds +1 to this cost. This inherited behaviour of compiler cost models, which by default penalizes inlining for code size reduction, obscures the exploration of novel and somehow counterintuitive alternatives such as the one proposed in this paper. Indeed, LLVM implements a heuristic to prioritize the inlining of solitary calls (see the `LastCallToStaticBonus` function at `InlineCost.cpp`). However, in contrast to our goals, as explained in Section 3.1, the goal of LLVM's heuristic is not to reduce code size. Rather, this heuristic exists in LLVM to limit code expansion. Thus, code is still expected to grow, albeit by a smaller factor.

6 Conclusion

This paper has presented a new perspective on inlining: instead of looking at it as a speed-oriented optimization, we showed that it is possible to use inlining also as a means for code size reduction. To this end, in Section 3 we have explored two inlining heuristics capable of reducing code size. One of them—rollback—departs from traditional implementations of inlining because it measures benefits a posteriori, after optimizations have trimmed the inlined code. Key to this approach is a combination of function cloning with traditional code-size optimizations to profit from context-sensitive information. Experiments in LLVM have shown that our heuristics could reduce the binary image of the MiBench programs compiled to x86 in 2.96%, while still keeping the performance improvements of standard inlining. In this paper, we have evaluated only two inlining heuristics tuned for code-size reduction. Nevertheless, the universe of potential techniques is much larger, and we hope that this study can inspire further explorations of this space.

Software. The implementation of the techniques described in this paper are available at https://github.com/vinicpac/llvm-project/tree/inline, under the GPL-3.0 license.

Acknowledgments

Fernando Pereira has been supported by CNPq (Grant 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt). The authors also thank PUC Minas and FAPEMIG for the scholarships that were made available to Thaís and Vinícius.

References

[1] Maia Ginsburg Andrew W. Appel. 1998. *Modern Compiler Implementation in C* (1 ed.). Cambridge University Press.

[2] Kim Bongjae, Yookun Cho, and Jiman Hong. 2012. An efficient function inlining scheme for resource-constrained embedded systems. *Journal of Information Science and Engineering* 28.

[3] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-size Optimization for Production iOS Mobile Applications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, US, 1–12.

[4] J. Bradley Chen and Bradley D. D. Leupen. 1997. Improving Instruction Locality with Just-in-Time Code Layout. In *Windows NT Workshop (NT'97)*. USENIX Association, USA, 4.

[5] Keith Cooper and Linda Torczon. 2012. *Engineering a compiler* (2 ed.). Elsevier.

[6] Ivica Crnkovic. 2005. Component-Based Software Engineering for Embedded Systems. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 712–713. https://doi.org/10.1145/1062455.1062631

[7] Anderson Faustino da Silva, Bruno Conde King, Jose Welsey Magalhaes, Breno Guimares, Jeronimo Nunes, and Fernando Magno Quintao Pereira. 2021. AnghaBench: a Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *CGO*. IEEE, USA, 378–390.

[8] Anderson Faustino da Silva, Bernardo Lima, and Fernando Magno Quintao Pereira. 2021. Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools. In *CC*. ACM, New York, NY, USA, 47–58.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *WWC*. IEEE, USA, 3–14.

[10] Masayo Haneda, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. 2006. Code Size Reduction by Compiler Tuning. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Stamatis Vassiliadis, Stephan Wong, and Timo D. Hämäläinen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 186–195.

[11] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and Incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, 111–121.

[12] Rainer Leupers and Peter Marwedel. 1999. Function inlining under code size constraints for embedded processors. In *1999 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 253–256.

[13] Guilherme Ottoni. 2018. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *PLDI*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3192366.3192374

[14] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. IEEE, New York, NY, USA, 233–244. http://dl.acm.org/citation.cfm?id=3049858

[15] Gabriel Poesia and Fernando Magno Quintão Pereira. 2020. Dynamic Dispatch of Context-Sensitive Optimizations. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428235

[16] A. Pohl, B. Cosenza, and B. Juurlink. 2018. Cost Modelling for Vectorization on ARM. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 644–645.

[17] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/1375581.1375609

[18] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, USA, 149–163.

[19] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868.

[20] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3377555.3377890

[21] P. W. Sathyanathan, W. He, and T. H. Tzen. 2017. Incremental whole program optimization and compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 221–232.

[22] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. 2018. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, 12:1–12:15.

[23] Zhou Xinrong, Lu Yan, and Johan Lilius. 2007. Function inlining in embedded systems with code size limitation. *The International Arab Journal of Information Technology* 2, 154–161.

[24] Peng Zhao and José N. Amaral. 2003. To inline or not to inline? Enhanced inlining decisions. In *LCPC*, Vol. 2958. 405–419.