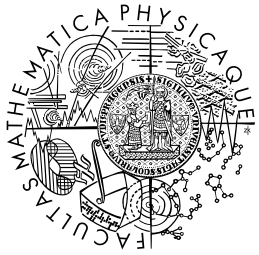

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Vojtěch Káně

Alternative inliner implementation in GNU Compiler Collection

Department of Applied Mathematics

Supervisor of the bachelor thesis: doc. Mgr. Jan Hubička, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank doc. Jan Hubička a lot for being so kind at all the consultations and for high flexibility despite tight deadlines.

I dedicate this work to the scouts of the 2nd and 5th scout group who took over my obligations to the group without hesitation and despite my predictions that these two activities will not interfere.

A special thanks go to my family and closest friends, who helped me to overcome this challenge. It was very helpful to have someone near to share my thoughts with. I would therefore like to express my grate gratitude to the circle of the S322 office, the community of Pátek, the scouts from my hometown and all other, who kindly offered advice or just warm words and who did not hesitate to hug me when I needed it.

I also thank the Department of Applied Mathematics for relaxing my work duties for a while and for providing me with computing power to properly test and benchmark my patches.

Děkuji moc doc. Janu Hubičkovi za vždy vstřícné konzultace a velkou flexibilitu navzdory blížícímu se termínu odevzdání.

Tuto práci věnuji skautům a skautkám ze Staré Boleslavi, kteří v posledních měsících bez váhání zastali mé povinnosti zejména ke 2. skautskému oddílu, aniž bych je na to předem připravil.

Zvláštní poděkování patří rodině a nejbližším, kteří mi pomohli tuto velkou výzvu překonat. Velkou zásluhu na vzniku této práce měli ti, kteří se mnou náročné chvíle „jen“ sdíleli. Touto formou bych proto rád vyjádřil velký vděk společenství pracovny S322, komunitě Pátek, skautům a skautkám mého dvojměstí a všem dalším, kteří neváhali přispěchat s radou, vlídným slovem či obejmutím.

Děkuji také Katedře aplikované matematiky za umožnění volnějšího pracovního režimu a za poskytnutí výpočetní infrastruktury pro otestování a proměření mnou navržených změn.

Title: Alternative inliner implementation in GNU Compiler Collection

Author: Vojtěch Káně

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Jan Hubička, Ph.D., Department of Applied Mathematics

Abstract: Inlining is a very important optimization pass of today's compilers. It saves function call overhead and provides more context for other optimization passes by replacing function's call site with its body. We revisit the current "greedy" inliner in GNU Compiler Collection, which was written more then 20 years ago and propose an alternative algorithm suitable for parallel processing. We combine the current approach of using a priority queue and the approach of the early inliner of traversing the callgraph in reverse post order by running the RPO traversal multiple times with increasing limits. Our measurements suggest the presented algorithm is worth further research and that properly tuning the constants may put it on a par with the current inliner all while allowing space for future parallelization of the IPA phase.

Keywords: compiler, inter-procedural optimization

Název práce: Alternativní implementace inlineru v GNU Compiler Collection

Autor: Vojtěch Káně

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: doc. Mgr. Jan Hubička, Ph.D., Katedra aplikované matematiky

Abstrakt: Inlinování je velmi důležitou součástí optimalizačních průchodů současných překladačů. Nahrazením volání funkce za její tělo se ušetří režie na provedení volání a získá více kontextu pro ostatní optimalizační průchody. V práci prozkoumáme současný „hladový“ inliner v GNU Compiler Collection napsaný před více než dvaceti lety a navrhneme alternativní algoritmus vhodný pro použití v paralelním prostředí. Spojíme současný přístup používající prioritní frontu a přístup early inlineru procházejícího funkce v topologickém pořadí (reverse post order) tím, že průchod provedeme opakovaně s postupně se zvyšujícími limity. Měření naznačují, že navržený algoritmus je hoden dalšího zkoumání a že nastavením vhodných konstant by se mohl postavit současnému inlineru a přesto být připravený na budoucí paralelizování IPA fáze překladu.

Klíčová slova: překladač, interprocedurální optimalizace

Contents

Introduction

GCC is one of the most used compilers of the C/C++ language in the world. It is well-known for being easily adaptable to new CPU architectures (so called back-ends) and new languages (front-ends). Despite this flexibility, it is capable of many advanced optimizations making it a common choice for CPU benchmarking. It was the only supported compiler by the Linux kernel until 2010. New features are contributed regularly including the recent initial support for the Rust and Go programming languages.

An important part of its success is the ability to do complex optimization, that is to produce a different program with equivalent observable behavior but a shorter runtime and/or smaller binary size. A precise and comprehensible definition of *observable behavior* is crucial as unexpected optimizations are a common source of bugs. The problem of perfect optimization is difficult from the theoretical point of view. Depending on the precise requirements, it is either unsolvable (Lemma 1) or NP-hard (Lemma 2).

Lemma 1. *Consider the RAM computational model as defined by Mareš&Valla in [1, Chapter 2.5]. The goal of* `opt` *is to take a program as input and transform it to one with equivalent observable behaviour and shortest possible runtime with the least amount of instructions. The only observable behavior is the final contents of the memory. No such* `opt` *can exist.*

Proof. Let `prog` be some program for RAM s. t. `opt`(`prog`) = `prog`. Consider the following program

Algorithm 1

1: Call `prog`
2: $A \leftarrow A + 1$ ▷ A is a memory cell

`opt` must omit the second line if and only if `prog` never terminates. The complexity of `opt` must therefore be at least the complexity of the halting problem, which is unsolvable [2]. □

Lemma 2. *Consider the RAM computational model as defined by Mareš&Valla in [1, Chapter 2.5]. The goal of* `opt` *is to take a program as input and transform it to one with equivalent observable behaviour and shortest possible runtime. The only observable behavior is the final contents of the memory.* `opt` *is NP-hard.*

Proof. Let `SAT` be some program for RAM that gets a logical formula of length n in memory cells A_1 to A_n and sets $A_0 = 1$ if the formula is satisfiable and sets $A_0 = 0$ otherwise. Consider the following program

n is a constant from the point of view of `SAT` so its runtime is trivially in $\mathcal{O}(1)$, but we are interested in the runtime of `opt` and it treats n as an input parameter.

Despite n being a constant for the SAT solver, it is the length of the input for `opt`. We know the output must look like

Every line sets the contents of distinct memory cells, so this program must be optimal. To obtain it, `opt` most select the correct variant of line 5 by solving the SAT problem. Mareš&Valla prove SAT to be NP-complete [1, Chapter 19.4]. `opt` is at lest as hard as SAT, therefore `opt` is NP-hard. □

Algorithm 2 SAT with constant input

1: $A_1 \leftarrow x_1$
2: $A_2 \leftarrow x_2$
3: $\vdots$
4: $A_n \leftarrow x_n$
5: Call `SAT`

Algorithm 3 Optimal `SAT` for constant input

1: $A_1 \leftarrow x_1$
2: $A_2 \leftarrow x_2$
3: $\vdots$
4: $A_n \leftarrow x_n$
5: $\begin{cases} A_0 \leftarrow 1, \text{if the formula is satisfiable} \\ A_0 \leftarrow 0, \text{otherwise} \end{cases}$

Given the lemmata above we will evaluate our proposed changes by experiments on real common hardware instead of relying on theoretical guarantees. We will also study the structure of ordinary computer programs and focus on them instead of the worst case scenarios.

1 GCC internals

We will describe the current state of GCC. We will introduce the process of compilation, the optimization passes involved and where they reside in the source code tree. We will primarily focus on inter-procedural optimization (IPA) and its inlining passes and describe the algorithm and data structures used.

Compilation of a single translation unit can be divided into three stages:

1. Parse the source code and destructure it to a common internal representation named GIMPLE [3, Chapter 12] (*front end*)

2. Perform common optimization and possibly other transformations including conversion to even simpler intermediary languages such as SSA form or RTL (*middle end*)

3. Output machine code for selected target (*back end*)

All of these stages perform optimization, but we try to concentrate it in middle end as much as possible so that all languages and all targets can benefit from it. The end-to-end compilation process is often more complicated including running the preprocessor and assembler but these steps are out of scope of this thesis.

An important property of a modern compiler is reproducibility of its output. If you compile the same source with the same options (including implicit ones), you must get the same output not only in terms of functionality, but really byte-by-byte identical binary. This makes it easier to reproduce and therefore fix bugs and more importantly makes the whole software supply chain more trustworthy. If the source and build process is known, one can verify whether given binaries are genuine. Special care must be taken while developing compilers to fulfil this requirement. All randomized algorithms must be seeded with a known constant or derived from input data and parallel algorithms must be thoroughly analyzed to prove they give the exact same results independent of the order of execution. As of today, GCC's compilation is fully reproducible. To make this property very well tested, GCC is compiled multiple times and mutually compared during a default bootstrap.

The middle end itself is composed of multiple phases each consisting of many passes. We will briefly list some of the passes, their full list can be found at [3, Chapter 9]. We present those we expect to be most positively affected by inlining. Most of them operate inside each function separately and therefore will benefit from the extended context caused by inlining.

- IPA passes
  - **inline** is the pass we want to modify
  - **constant propagation** checks what arguments are passed to functions and whether their bodies can be simplified using this observation
- tree SSA passes
  - **dead code elimination** drops statements without side effects and with unused results

  - **forward store motion** moves assignments closer to their use point
  - **unrolling of small loops** completely unrolls loops with few iterations
  - **dead store elimination** removes stores to memory that are never used

- RTL passes
  - **common subexpression elimination** merges redundant computations inside basic blocks
  - **loop optimization** is a larger collection of transformations including loop invariant motions, unrolling and peeling

Most of these passes are well-known and time-tested. We may get the impression they are very cheap in terms of computing power since they are old and computers got orders of magnitude faster in the meantime. Not only do we have to account for code side increase as well, an important game changer was the introduction of LTO to GCC in 2009 [4]. Link time optimization is the process of running interprocedural optimization passes across translation unit boundaries. This brings important improvements to both speed and code size, but puts extreme pressure on IPA passes to be performant and not too memory hungry. Despite having *link time* in its name, the optimization can be done neither by a regular linker nor with regular object files. Machine code lacks necessary metadata to make the optimization possible. GCC solves this problem by dumping GIMPLE serialization to the object file and replacing the linker with almost the full compiler [3, Chapter 25]. In the high level overview, this resembles splitting the compiler around the middle-end dumping the output to a file and then later continuing.

Inlining is the process of replacing a function call with the callee's body. This can save a couple of instructions of code size if the callee is short, saves a couple of instructions of overhead for the call itself, but most importantly it allows for other optimization passes to provide better results. This is especially important for languages with support for generic programming. The current inliner in GCC was initially written and continuously improved by Jan Hubička in 2003 [5, `gcc/ipa-inline.cc`]. It operates on a callgraph, which is a directed multi-graph with loops. Nodes represent functions and edges point from caller to callee. They are stored as a list of pointers to neighbors [5, `gcc/cgraph.h`]. The callgraph is suited for single-threaded access only at the time of writing, but we are not aware of any obstacle in adding a lock to each node other than having to repetitively modify large parts of the compiler. If lock access collisions are infrequent, we expect the performance cost to be negligible. It will still require some thought to cope with the reproducibility requirements mentioned earlier, but still there is no fundamental obstacle.

During the inlining phases, functions have already been partly optimized and their summaries have been generated. There are two subpasses; `pass_early_inlining` and `pass_ipa_inline`. Both of them share a large amount of helper code [5, `gcc/ipa-inline.cc`] [5, `gcc/ipa-utils.cc`] [5, `gcc/ipa-inline-transform.cc`] [5, `gcc/ipa-inline-analysis.cc`]. It makes their algorithms easy to reason about yet capable of complex and powerful decisions. One of the

most important utility function is estimating edge growth and computing edge badness.

Edge growth tells us the increase (or decrease in case of negative growth) in code size in case we inline the selected edge. It is an integer representing the change in an amount of instructions. The problem is IPA is part of the middle-end and as such does not operate on instructions. We do not even know the target architecture in this phase. All calculations are therefore approximated with idealized instructions derived from the intermediary language. It tends to overestimate the size a little as it can not account for target specific SIMD instructions. We could not prove however that it does not underestimate in special cases.

Badness is a combined metric to compare edges by the usefulness of their inlining. It is a real number represented by GCC's own `sreal` data type. Its unusual sign orientation was chosen so that it fits the usual variant of heaps – minimal heaps. The central idea is to compute a speed/size trade-off as

$$-\frac{\text{instructions saved} \cdot \text{execution count of caller}}{\text{growth of caller} \cdot \text{overall growth} \cdot \text{combined size}}$$

with operands estimated from the profile. Dynamic profile (a measurement of runtime characteristics) is preferable and it makes badness a global property of each function. If it is not available, static profile (a prediction of the runtime characteristics) is used. That makes badness a property of each edge as it tries to estimate the importance by looking at the surroundings of the functions call. It is also possible to explicitly disable the static profile and base badness on the depth of nested loops in callee (so called *loop nest*) and the overall size growth. In each case, the base badness is then adjusted multiple times to account for other signals as well. We will provide a couple of such modifications, the full list can be found at the source code of the `edge_badness` function [5, `gcc/ipa-inline.cc`].

- strongly prefer edges with non-positive growth
- prefer functions with little callers (if it gets inlined to all of them, it can be excluded from the binary)
- prefer edges that turn indirect calls to directs ones when inlined
- prefer functions declared *inline*

1.1 Early inliner

Early inlining is focused on obviously beneficial cases only therefore being algorithmically simple and performant. Its goal is to eliminate high abstraction penalty in C++ code and possibly other similar languages like Rust [6]. It traverses the callgraph in reverse post order ("bottom up"). Functions have already been optimized in this order of traversal, so more of them appear as good-fit for inlining. The decision heuristics are very simple, namely:

- inline functions with `always_inline` attribute

- recursively inline functions with `flatten` attribute
- inline so small functions, that inlining them decreases total code size
- inline so small leaf functions, that inlining them increases the total code size by at most `early_inlining_insns` (it is believed this tiny penalty will later disappear thanks to other optimizations)
- skip recursive edges

1.2 Greedy inliner

The full `pass_ipa_inline` does a generic traversal called `inline_small_functions` and then some special-case transformations.

Algorithm 4 Pseudocode of `inline_small_functions`

```
1: E ← all edges reachable from exported nodes
2: while unit growth wasn't reached and E ≠ ∅ do
3:     e ← edge with lowest badness from E
4:     if e is inlinable & inlining e is profitable then
5:         Inline e
6:         Recursively update summaries of callers of e
7:         Add newly discovered edges to E [a]
8:     end if
9:     E ← E \ e
10: end while
```

[a] inlining may turn indirect edges to direct ones

To make updates of E fast, it is backed by a fibonacci heap [1, Chapter 18.4]. It is a fast heap both in theory and as used in GCC, but it can not be used in SMP scenarios directly. While there exist attempts to modify fibonacci heaps for SMP [7], we believe avoiding it completely is a better long-term choice improving both performance and comprehensibility.

Decisions on line 4 are mostly governed by `can_inline_edge_p` and `want_inline_small_function_p`. Recursive edges are still skipped. For each callee, it is checked whether it's size is not too large and also whether the unit won't grow above `param_inline_unit_growth` after the inlining operation. The parameter is local for each function, so no short-circuiting of the loop is possible. Given the priority queue E, it is equally important to have a low `badness` score to get inlined.

While there are many special tweaks, the main idea of `badness` is to prefer small (ideally negative) code size increase and to prefer hot edges (those called frequently). This approach is common in compiler design [8, Chapter 1].

An edge is marked for inlining by calling `inline_call` [5, `gcc/ipa-inline-transform.cc`]. The transformation is not done right away, the edge is marked and all are processed later. We need to push this information to the callee as well not to confuse later IPA stages (unused unexported functions shall be removed for example). This is done by creating a so-called inline clone. This implementation

detail is not important for the current inliner, but it will have to be handled by our proposed solution.

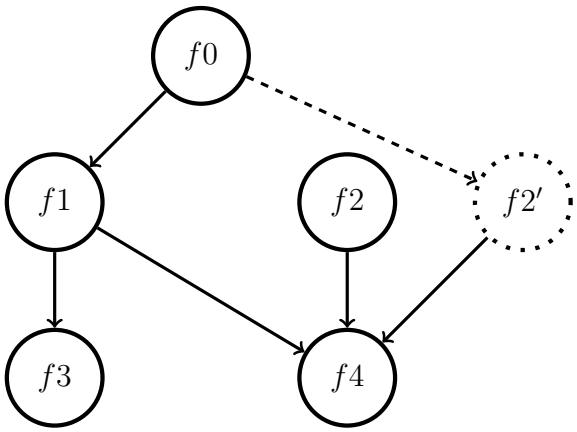

Figure 1.1 Callgraph after inlining $(f0, f2')$ edge

2 The naive inliner

We propose a different implementation of the `inline_small_functions` of the `pass_ipa_inline`. We want to achieve at least the speed of the current implementation, deliver optimized code with not much worse speed/size ratio and use only SMP-friendly algorithms and data structures. As stated in chapter 1, the callgraph is not prepared for concurrent access, so we can not bring any immediate performance improvements over the current inliner. The goal of the thesis is therefore to make some first steps in making the inlining phase parallel and also to prove that making the necessary changes for concurrent access to the callgraph is justifiable.

Our approach is led by the intuition of where most of the hot code resides in the callgraph and by observations of common C++ abstraction patterns. Because of relying on such non-exact hints, we call our result the *naive inliner*. We are aware of a similar concept being used in the LLVM [9, `mlir/lib/Transforms/Utils/Inliner.cpp`]. That itself is not a guarantee of positive results as the overall result largely depends on other optimization passes (including GCC's `inline_small_functions`) and how exactly the profitability of inlining each edge is approximated. We will therefore provide our own measurements.

The central part of the naive inliner is

Algorithm 5 Naive `inline_small_functions`

1: $F \leftarrow$ all functions sorted in reverse post order
2: **for** $f \leftarrow F$ **do** ▷ loop over all elements of F
3: **for** $e \leftarrow \{e | e \in f_{callees} \& \neg e \text{ inlined}\}$ **do** ▷ loop over noninlined calls in f
4: **if** e is inlinable and profitable & growth of $e \leq \text{thresh}_g$ & badness of $e \leq \text{thresh}_b$ **then**
5: Inline e
6: Update summaries of caller of e
7: **end if**
8: **end for**
9: **for** $e \leftarrow \{e | e \in f_{callees} \& e \text{ inlined}\}$ **do** ▷ loop over inlined calls in f
10: Call `inline_inlined`(e)
11: **end for**
12: **end for**

We iterate over the callgraph bottom-up and inline all "good-enough" edges that do not inflate the program too much. To keep the promise of easy parallelization, both thresh_g and thresh_b must be constant during naive `inline_small_functions`' execution. We no longer have the guarantee of keeping the total growth strictly bellow limits as there exists adversarial programs that can grow almost arbitrarily. As an example, consider a program with all edge growths equal to γ. By varying the thresh_g either no edges get inlined (hurting resulting performance) or all edges do (increasing the program size by $\gamma \cdot \sum |f_{callees}|$). While it may be possible to come up with values of the thresholds such that real-world scenarios behave reasonably, predicting them is very difficult since properties of functions change as other ones are inlined into them.

Directly accounting the modifications described above would bring us back to the current implementation – precise but nonparallelizable. We choose a different path and run the `inline_small_functions` multiple times called rounds. We schedule a single thresh_g for all rounds, but we keep slowly increasing thresh_b. Adversarial programs still exist (optimal inlining is NP-complete [10]), but we can stop soon-enough for sane cases by simply not running another round. Another important benefit is however in revisiting skipped nodes. This simulates the original priority queue without requiring synchronization.

Unlike the current inliner that collects all edges in advance and later traverses them, the naive inliner walks through functions collected in advance. If more functions are created during a round (e. g. inline clones) they will not be considered in later rounds with looser thresholds. Take for example Figure 1.1 in page 12. Edge $(f0, f2')$ was inlined during round 1. After rising thresholds for round 2, edge $(f2, f4)$ may become acceptable for inlining. Doing so is pointless however as $f2$ is effectively unused. More importantly, we will never visit $f2'$ to learn about $(f2', f4)$ to consider it for inlining. This is the problem we try to solve with `inline_inlined`.

Algorithm 6 `inline_inlined`(e)

1: **if** callee of e is marked as inlined **then**
2: Call `inline_inlined`(`callee of` e)
3: **else**
4: Consider e for inlining the same way naive `inline_small_functions` does
5: **end if**

In the language of Figure 1.1, when we see a dashed edge, we start a depth first search for non-dashed ones ($f0$ might have gotten inlined in the meantime for example) and consider those for inlining.

It remains to decide the correct values of thresh_g and thresh_b. GCC currently has a flag `inline-unit-growth` to specify the maximum allowed growth factor for each translation unit. It could then be applied as

$$\texttt{current\_size} + \texttt{edge\_growth} \leq \texttt{initial\_size} \cdot \texttt{inline-unit-growth}.$$

`inline-unit-growth` can be adjusted per each function, but it does not matter for the above formula. The real problem is with maintaining up-to-date value of `current_size` in a concurrent computation. We therefore use the formula

$$\texttt{size\_at\_round\_start} + \texttt{edge\_growth} \leq \texttt{initial\_size} \cdot \texttt{inline-unit-growth}.$$

This way we only guard against extreme momentary growths. The total one will have to be protected by having short rounds therefore making `size_at_round_start` reasonably accurate. What remains for controlling round size is thresh_b. We did some experimenting with picking its values statically based on the structure of some reference programs, but it turned out the distribution of `badness` is far from uniform as can be seen in Figure 2.1.

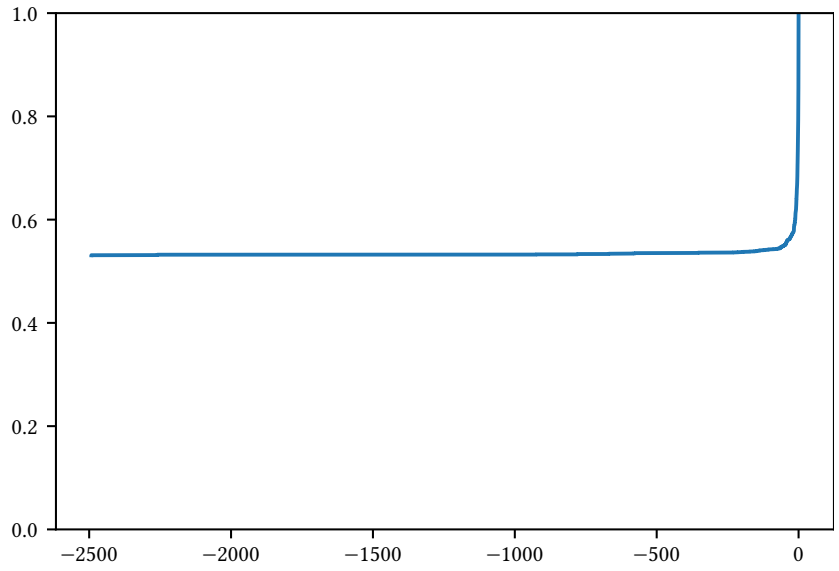

Figure 2.1 Distribution of `badness` in Tramp3Dv4 [11]. x-axis shows values of *badness*, the function is $f(b) = \frac{\#\text{ of functions with } \texttt{badness} \leq b}{\#\text{ number of functions}}$. `badness` of $-\infty$ is omitted.

3 Experimental results

An important part of altering a compiler is justifying the change. Led by the observation that execution the actual inlining transformations is much slower than planning them, we will concentrate instead on the speed of the resulting binaries. We have confirmed this expectation by compiling clang `??` with both inliners and comparing the length of the inlining phase as output by the `-Q` option. Another reason for this approach is the fact, that the main performance benefits will be unleashed by parallelization of the rounds, which is far out of the scope of this theses as explained earlier.

The SPEC benchmark is exceptional for this thesis as it is not freely available and had to be executed by the supervisor. Its details are mentioned in its subsection. All other benchmarks were executed on a common HW, namely a computer with:

- **CPU** AMD Ryzen 5 7600 (6 cores ×2 hyper-threading, 3.8 GHz base, up to 5.1 GHz)
- **RAM** consisting of 2 modules of Crucial Pro 16 GB DDR5-5600 (DDR5, 46-45-45)
- **storage** Samsung SSD 970 EVO Plus 1TB
- **Linux kernel** identifying itself as `6.1.0-21-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.90-1 (2024-05-03) x86_64 GNU/Linux`
- **virtualisation** was not used

We want to avoid synthetic measurements and therefore we try to capture events observable by end user (e. g. total execution time) in environments used by end users (commodity HW, common kernel version…). Each test was executed multiple times to counter hard-to-control reproducibility issues (the state of CPU caches for example). We call each test execution a *run* and a full series of runs a *batch*. The default batch size was 10. The kernel page cache was emptied before each batch with `sync && echo 3 > /proc/sys/vm/drop_caches`. It was intentionally left out between individual runs as syscall patterns can be affected by inlining and therefore the performance difference is relevant. It is also true that caches are used during normal execution.

Whenever we measure the size of a binary, we concentrate on its `.text` section only. This is to exclude debugging symbols and other contents other contents not related to the actual code size.

3.1 SPEC

SPEC CPU® 2017 by Standard Performance Evaluation Corporation is an industry standard benchmark to evaluate performance of both CPUs and compilers [12] [13]. Its source code is not freely available, but it is otherwise well known to compiler designers so that is why we present it here. The tests were executed by the supervisor on a machine equipped with AMD EPYC CPU. Table 3.1

and Table 3.2 list the difference of the original inliner and the naive one for each test.

The Imagemagick benchmark is known to be sensitive to inlining for reasons that are not understood by the inlining heuristics. In particular, when certain functions are not inlined, data-structures containing RGBA data are stored by pieces in the caller and loaded as a vector in the callee. This causes CPU to perform long penalty on mismatches between sizes of loads and stores. The Exchange benchmark is producing difficult sudoku tables. It is written as a self-recursive function which recurses at most 10 times. With `-Ofast` GCC produces 10 clones of this function and the overall performance is highly sensitive on how individual inline decisions corellate with the tree of recursion which is unknown to the compiler.

Other performance results are pretty close, especially for the `-O2` variant. We consider the isolated performance drops not a fundamental flaw of the naive inliner's idea and we believe it can be reduced with further investigation and tuning of the constants driving the inliner's decisions. The size results are a bit worse with up to $\frac{1}{4}\times$ increase. This is caused by the fact the naive inliner does not measure the accumulated size increase precisely and therefore tends to overgrow the set limit. There is a natural question of how this overgrowing relates to the number of rounds. We will elaborate on it in the analysis of Tramp in Section 3.2.

3.2 Tramp3Dv4

Tramp3Dv4 is a program computing extensive physical simulations [11]. It is written in C++ in a very high-level way (we say there is a high *abstraction penalty*) which is why it is commonly used to benchmark compilers and especially their ability of inlining as missing an important opportunity is immediately visible [6] [14]. It is also distributed as a single source file and produces an easy to run binary without additional input data making the benchmarking simple. During the development of the naive inliner, it was observed that disabling the IPA inline small functions completely has little to no effect. This is caused by the early inliner, which specializes at removing C++'s abstraction penalty, as stated in Chapter 1. We will therefore provide some batches with the early inliner disabled. The results given are an average of the batch unless stated otherwise.

Tramp3Dv4 was compiled with the following command line (and optionally `-fipa-naive-inline`)

```
g++ -fpermissive -O3 -march=native -fno-early-inlining
    tramp3d-v4.cpp
```

The source code had to be modified by adding

```
using std::isinf;
using std::isnan;
```

right under the block of `#include`s to accommodate to changes in the language and its standard library. This is also the cause of the `-fpermissive` option. To utilize the extended context for other optimizations caused by inlining, we employed the `-O3` and `-march=native` options.

Test name	Rate `-O2`	Rate `-Ofast`	Size `-O2`	Size `-Ofast`
503.bwaves_r	0.22 %	0.25 %	0.00 %	0.00 %
507.cactuBSSN_r	−1.21 %	−2.99 %	0.16 %	1.76 %
508.namd_r	−3.18 %	0.92 %	0.00 %	11.34 %
510.parest_r	3.05 %	0.71 %	7.96 %	11.55 %
511.povray_r	−6.85 %	−10.00 %	−2.73 %	13.14 %
519.lbm_r	−0.95 %	−1.31 %	0.20 %	−0.87 %
521.wrf_r	1.71 %	1.23 %	0.01 %	−0.03 %
526.blender_r	−2.04 %	0.52 %	3.10 %	16.72 %
527.cam4_r	−0.24 %	0.88 %	0.04 %	−0.18 %
538.imagick_r	−0.37 %	37.03 %	2.27 %	11.34 %
544.nab_r	0.14 %	−0.12 %	0.00 %	3.03 %
549.fotonik3d_r	−0.46 %	0.38 %	0.00 %	0.00 %
554.roms_r	−1.56 %	0.00 %	0.00 %	0.09 %
Geometric average	−0.92 %	1.56 %		
Arithmetic average			0.85 %	5.22 %

Table 3.1 Relative rate differences of the SPEC CPU 2017 floating point benchmark for different optimization flag sets taking a median of three evaluations. Positive numbers mean the original inliner provided better results.

Test name	Rate `-O2`	Rate `-Ofast`	Size `-O2`	Size `-Ofast`
500.perlbench_r	−3.68 %	3.39 %	−5.07 %	11.90 %
502.gcc_r	0.86 %	1.78 %	5.52 %	15.50 %
505.mcf_r	1.23 %	−0.72 %	−2.71 %	−0.71 %
520.omnetpp_r	−1.03 %	4.77 %	10.36 %	19.02 %
523.xalancbmk_r	7.01 %	2.68 %	15.30 %	12.52 %
525.x264_r	−0.35 %	0.85 %	2.85 %	3.34 %
531.deepsjeng_r	0.40 %	0.00 %	0.81 %	−0.86 %
541.leela_r	1.90 %	1.75 %	3.16 %	17.44 %
548.exchange2_r	0.00 %	8.47 %	0.00 %	−0.01 %
557.xz_r	0.53 %	5.74 %	23.22 %	17.01 %
Geometric average	0.67 %	2.75 %		
Arithmetic average			5.34 %	9.52 %

Table 3.2 Relative rate and size differences of the SPEC CPU 2017 integer benchmark for different optimization flag sets taking a median of three evaluations. Positive numbers mean the original inliner provided better results.

We opted for the runtime options provided by Tramp's help and only requested a particular number of iterations by

```
./tramp3d-v4 --cartvis 1.0 0.0 --rhomin 1e-8 -n 100
```

	Size	**Runtime**
original inliner	581776 B	37.08 s
naive inliner	593168 B	2.49 s
relative difference	2 %	-1489 %

Table 3.3 Results of benchmarking Trump3Dv4 with early inliner disabled

The orders of magnitude large difference in Table 3.3 is caused by the absence of early inlining. We will therefore redo the test with `-fno-early-inlining` removed for the original inliner only.

	Size	**Runtime**
original inliner	649310 B	2.48 s
naive inliner	593168 B	2.49 s
relative difference	-9.5 %	0.4 %

Table 3.4 Results of benchmarking Trump3Dv4 with early inliner disabled for the naive variant

The combination of Table 3.3 and Table 3.4 may indicate that the naive inliner replaces the early inliner in certain circumstances and that the reverse post order traversal is a better overall choice.

We also asked ourselves how the behavior of both inliners is affected by the limits, most notably `inline-unit-growth`. We therefore measured the speed and size of Tramp3Dv4 with varying this parameter and keeping the early inliner on for the original inliner only. The limits of 1000 and 10000 produce identical binaries proving that they act as local ∞ and that the inliner was limited by `can_inline_edge_p` only.

The results of Figure 3.2 further hint us that the crucial part of removing abstraction penalty is the reverse post order, not the continuous execution of other optimizations. We will investigate whether this generalizes to speeding up programs with little abstraction penalty when benchmarking GCC in Section 3.4.

The combination of Figure 3.1 and Figure 3.2 nicely highlight that running inliners with arbitrarily high limits causes binary sizes explosion without justifiable performance gains. Both inliners begin inlining rather cold edges at approximately the same limits. It is also interesting to observe the behavior of the original inliner, which clearly misses a single important edge (or possibly all calls of a single function) for a while and then finally gets the budget to realize it (see drops at limit 10 and 50). The edge however doesn't cause any size increase if inlined in different order as can be seen at the naive inliner with limit 0. We will try to find out the offending function by inspecting the runtime profile by the Linux specific tool *perf* right before the drop and right after it, see Table 3.5 and Table 3.6. The symbol names consist of the full template argument assignment making it with up

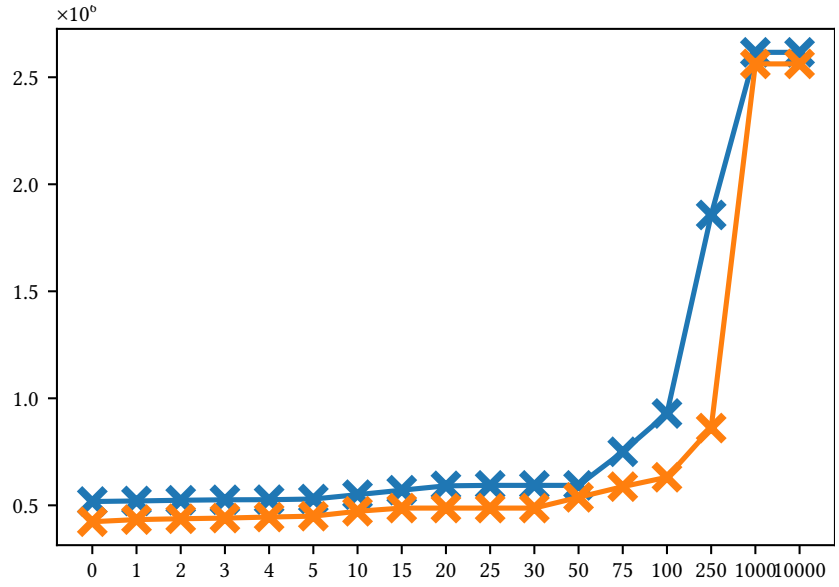

Figure 3.1 Size of `.text` section of Tramp3Dv4 in millions of bytes based on the value of `inline-unit-growth` parameter. Orange line corresponds to the original inliner, blue line to the naive one.

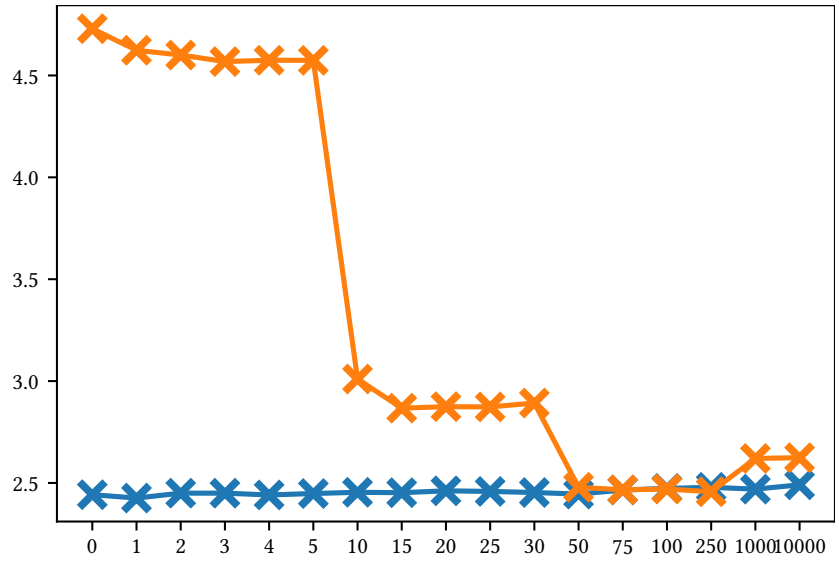

Figure 3.2 Duration of execution of Tramp3Dv4 in seconds based on the value of `inline-unit-growth` parameter. Orange line corresponds to the original inliner, blue line to the naive one.

Overhead	Symbol
9.61 %	`UniformRectilinearMesh<…>::cellPosition`
5.82 %	`UniformRectilinearMesh<…>::cellPosition`
4.82 %	`UniformRectilinearMesh<…>::vertexPosition`
4.33 %	`MultiArgKernel<…>::run`
4.19 %	`KernelEvaluator<…>::evaluate<…>`

Table 3.5 Top 5 functions of Tramp3Dv4 with `inline-unit-growth=5`

Overhead	Symbol
6.17 %	`KernelEvaluator<…>::evaluate<>`
5.43 %	`KernelEvaluator<…>::evaluate<>`
4.58 %	`MultiArgKernel<…>::run`
4.21 %	`MultiArgKernel<…>::run`
4.00 %	`MultiArgKernel<…>::run`
3.89 %	`MultiArgKernel<…>::run`

Table 3.6 Top 5 functions of Tramp3Dv4 with `inline-unit-growth=10`

to 900 characters unsuitable for typesetting on a page. We therefore provide very limited overview with all template arguments left out to illustrate the method and direct readers with further interest to attachments in Section A.2.

We also see the tendencies of the naive inliner to grow the binary slightly above the set limits. While we get a general idea by measuring the difference in Figure 3.1, it can not be used for exact calculations. The limit is specified relative to the normal binary size and both algorithms differ slightly in the accounting method. The original inliner relates it to the minimum size achieved until now, so inlining edges with negative growth makes the absolute value of the limit smaller. The naive inliner can not keep track of the minimum size and therefore relates everything to the initial one at the beginning of the phase. This is especially observable when using `inline-unit-growth=0`, because the naive inliner will first inline edges with non-positive growth thus buying itself some size reserve and then inlines even growing edges.

3.3 OpenSCAD

OpenSCAD is a computer aided design software used to produce 3D models based on declarative textual program description [15]. It was selected as an example of a moderately complex software which is widely known in the technical community. Thanks to its command line interface, it is possible to exactly measure the performance of its core functionality, which is an otherwise complex problem for most user-facing software [16]. We will task the software with materializing the geometry of a conceptually simple shape, we will however request high precision of the operation. The model is as follows

```
$fn=200;
size=100;

difference() {
```

```
        cube(size);
        sphere(r=size);
}
```

OpenSCAD is an extensive software yet it utilizes some external libraries, which may prove critical for the overall performance. We will therefore have to recompile them with the naive inliner as well. It would be easy to make mistakes in this process especially for libraries used by other libraries, so we will use the Nix build system [17] [18] to rebuild all transitive dependencies with the modified GCC as proposed in Chapter 2. To enable the naive inliner as reliably as possible we alter the source to make it the default choice. We otherwise base the build on nixpkgs' version current at the time of writing, which is `release-24.05` at commit `0f8e5a078ae2ba3c07424c7d7312d7bd11ad7037`. To give both inliners as many opportunities as possible, we enable LTO for the build. This is difficult to do globally as a lot of software fails to compile if you simply pass `-flto` to the build system. We will therefore do so for OpenSCAD and its direct dependencies which we consider performance critical, namely *eigen*, *opencsg*, *cgal_4* and *boost*.

We simply instruct OpenSCAD to materialize the above model and output it as textual STL by

```
openscad -o output.stl model.scad
```

	Size	**Runtime**
original inliner	5706519 B	7.57 s
naive inliner	5852919 B	7.71 s
relative difference	2.6 %	1.8 %

Table 3.7 Results of benchmarking OpenSCAD

The results in Table 3.7 fit the general tendencies set by other benchmarks.

3.4 GCC

The details of this software is already described in Chapter 1. We use it as an example of a complex software with large codebase but low abstraction penalty. We have already shown in Section 3.2 that the naive inliner excels in removing high abstraction penalty. We will make conceptually similar measurements on GCC to observe the behavior on a rather conservative source code.

We do the full boostrap of the compiler in the version proposed in Chapter **??** with LTO enabled and with varying the `inline-unit-growth` parameter. The following snippet starts the build assuming there is a `src` folder with GCC's source code in process working directory

```
mkdir build && cd build
../src/configure --with-build-config=bootstrap-lto
make BOOT_CFLAGS="-O2 -g --param
   inline-unit-growth=<growth> bootstrap
```

We will benchmark each compiler by simply using it to compile Tramp3Dv4 with the same options as in Section 3.2 with the naive inliner. To stay consistent

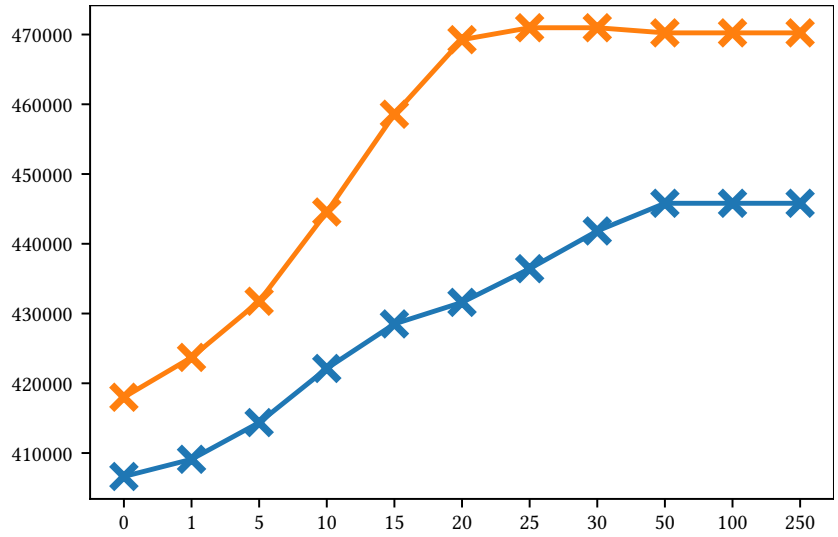

Figure 3.3 Size of `.text` section of GCC [5] in bytes based on the value of `inline-unit-growth` parameter. Orange line corresponds to the original inliner, blue line to the naive one.

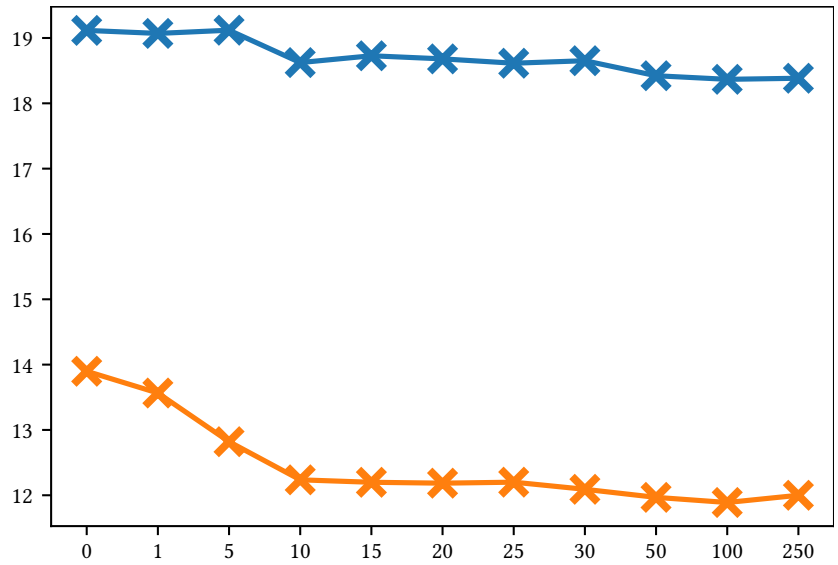

Figure 3.4 Duration in seconds of compilation of Tramp3Dv4 using GCC [5] with varying value of `inline-unit-growth` parameter. Orange line corresponds to the original inliner, blue line to the naive one.

with other benchmarks, we will measure the performance with the `time` command despite GCC having its own `-ftime-report`.

Both inliners seem to exhibit similar size change when the `inline-unit-growth` parameter grows, however the naive one always produces a smaller binary (Figure **??**). For performance, this relation is reversed – both behave similarly, but the original one is always faster (Figure **??**).

3.5 clang

We also decided to inspect the clang compiler of the LLVM suite [9]. It is a compiler of similar scope as GCC, but it is much younger, which can be seen on the source code. It utilizes more C++ features accounting for a higher abstraction penalty while being written with performance in mind. As with GCC, we benchmark it by letting it compile the Tramp3Dv4 project. We were unfortunately unable to make the compiler work. LLVM is known for targeting itself only, so its code exhibits undefined behavior from time to time when compiler by a different compiler. We therefore used it with the `-fsyntax-only` flag to only run the parser.

This benchmark was run on the same machine as the SPEC suite (Section 3.1) so only relative data are relevant.

Size	**Runtime**
-11.24 %	21.07 %

Table 3.8 Relative results of benchmarking clang. Positive values mean the original inliner scored better.

The results in Table 3.8 agree with the observations about GCC (Section 3.4). The naive inliner produces significantly smaller binaries with significantly large runtime. This applies however to large programs only.

4 User guide

GCC is a developer oriented software and as such does not provide any graphical user interface by default. The command line interface is rich and thoroughly documented in various formats [5, `gcc/doc`]. We do not intend to reproduce it here as that would not provide any new value to the reader.

The only new option added is `-fipa-naive-inline`. When used, it replaces the standard IPA `inline_small_functions` with the naive algorithm. It still honors the `inline-unit-growth` parameter and other relevant ones.

Our patches are not part of the official repository at the time of writing, so they can be found in Attachment A.1 with instruction how to apply to the source tree. The compiler can then by built with similar techniques as during the benchmarking, that is (assuming the modified source code is in `src` subdirectory of the process working directory)

```
mkdir build && cd build
../src/configure
make bootstrap
```

It is also possible to enable LTO for the build by passing `--with-build-config=bootstrap-lto` to the `configure` script (this does not affect the ability of the resulting compiler to do LTO). Very large time savings can be obtained on common hardware when running multiple threads of the build with `-j<n>`.

The resulting executables can be found at `build/gcc/xgcc` and `build/gcc/xg++`, but it can be difficult to use them because of their assumptions on the environment. One can instead install it to an empty destination with

```
cd build
make DESTDIR=/path/to/destination install
```

and use `/path/to/destination/usr/local/bin/gcc` without issues.

Conclusion

We present an implementation of an alternative inliner algorithm (the *naive* inliner) in GNU Compiler Collection. Its goal is to allow for easy parallelization after modifying other parts of the compiler while not being too much worse than the current algorithm. The naive inliner traverses the callgraph in reverse post order instead and tries to prioritize most profitable edges by running such a round repeatedly, with looser limits each time. It was argued that a single round can be processed by multiple threads at the same time thus fulfilling the goal of parallelizability. We then present numerous benchmarks inspecting performance and size of the resulting executables. The overall results suggest that the naive inliner consistently produces slightly larger binaries and that the performance is slightly worse in most cases. We believe this meets the set goals and consider it a success in this early stage of exploring the idea.

Apart of the overall comparison, there are also numerous interesting characteristics of the naive inliner. It seems to perform better on small and medium-sized programs (compare Section 3.3 with Section 3.5). On the other hand, it handles the abstraction penalty well (see code size decrease in Section 3.5) and can even replace the early inliner (see Section 3.2). Section 3.2 even suggests it excels with low `inline-unit-growth` parameter and only gets outrun by the original inliner with larger values of the limit. This statement however needs further investigation as the interpretation of the limit differs between the inliners.

We have shown the bottom-up traversal provides interesting results and we believe it shall be researched further. We propose to investigate the influence of the number of rounds in future works as the current value of 10 was chosen rather arbitrarily. It would also be made a configurable parameter of the compiler and possible have different default values for different optimization levels.

Bibliography

1. Mareš, Martin; Valla, Tomáš. *Průvodce labyrintem algoritmů*. Průvodce labyrintem algoritmů. 2. vydání. Praha: CZ.NIC, 2022. CZ.NIC. ISBN 978-80-88168-63-8.
2. Turing, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. 1937, vol. s2-42, no. 1, pp. 230–265. ISSN 0024-6115.
3. Stallman, Richard M.; GCC Developer Community. *GNU Compiler Collection Internals*. 2024. Available at gcc/doc/gccint.pdf at the GCC repository [5].
4. Novillo, Diego. *LTO branch merged into trunk - trunk remains CLOSED*. 2009. Available also from: `https://gcc.gnu.org/legacy-ml/gcc/2009-10/msg00060.html`. Message-ID: <20091003211217.GA22755@google.com>.
5. Free Software Foundation, Inc. *GNU Compiler Collection*. 2024. Available also from: `git://gcc.gnu.org/git/gcc.git`. Commit cbed07845708f01a122e02016660a9152e5c14ff.
6. Hubička, Jan. Interprocedural optimization on function local SSA form in GCC. In: *Proceedings of the GCC Developers' Summit*. Ottawa, Ontario, Canada, 2006.
7. Bhattarai, Divya. *Towards Scalable Parallel Fibonacci Heap Implementation*. 2018. Available at `https://repository.stcloudstate.edu/csit_etds/24/`.
8. Zhao, Peng; Amaral, José Nelson. To inline or not to inline? Enhanced inlining decisions. In: *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16*. Springer, 2004, pp. 405–419.
9. LLVM Team. *LLVM project*. 2024. Available also from: `https://github.com/llvm/llvm-project.git`. Commit 6009708b4367171ccdbf4b5905cb6a803753fe18.
10. Scheifler, Robert W. An analysis of inline substitution for a structured programming language. *Communications of the ACM*. 1977, vol. 20, no. 9, pp. 647–654.
11. Günther, Richard. *Three-dimensional parallel hydrodynamics and astrophysical applications*. 2005. PhD thesis. Universität Tübingen.
12. *SPEC CPU® 2017* [online]. [N.d.]. [visited on 2024-07-16]. Available from: `https://www.spec.org/cpu2017/`.
13. Limaye, Ankur; Adegbija, Tosiron. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018, pp. 149–158. Available from DOI: `10.1109/ISPASS.2018.00028`.
14. Jambor, Martin. Interprocedural optimizations of function parameters. *Proceedings of the 2009 GCC Developers' Summit*. 2009, pp. 53–64.

15. *OpenSCAD – The Programmers Solid 3D CAD Modeller* [online]. [N.d.]. [visited on 2024-07-16]. Available from: `https://openscad.org`.

16. Nielson, Jordan; Williamson, Carey; Arlitt, Martin. Benchmarking modern web browsers. In: *2nd IEEE Workshop on Hot Topics in Web Systems and Technologies*. Citeseer, 2008.

17. Dolstra, Eelco. *The Purely Functional Software Deployment Model*. 2006. PhD thesis. Utrecht University. Available at `https://edolstra.github.io/pubs/phd-thesis.pdf`.

18. *Nix Reference Manual* [online]. [N.d.]. [visited on 2024-07-16]. Available from: `https://nix.dev/manual/nix/2.18`.

List of Abbreviations

- **CPU** – *central processing unit*
- **GCC** – *GNU Compiler Collection*
- **HW** – *hardware*
- **IPA** – *interprocedural analysis*
- **LLVM** – despite the capitalization, this is not an abbreviation, it is the full name of a compiler related project
- **LTO** – *link time optimization*
- **RAM** – *random access memory* or *random access machine*, intended expansion shall be clear from the context of usage
- **RTL** – *register transfer language*
- **SAT** – *satisfiability*
- **SIMD** – *single instruction multiple data*
- **SMP** – *symmetric multiprocessing*
- **SSA** – *single static assignment*
- **STL** – *stereolithography*

A Attachments

A.1 Patches

The output of this thesis is a modification of GCC's source code. It is attached in electronic form as multiple patches based on the version cited throughout this paper [5]. The file `naive-inline.patch` contains the actual inliner as described in Chapter 2. Details on how to use it can be found in Chapter 4. The file `fixes.patch` contains modifications irrelevant to the main goal of this thesis. None of them is known to affect the behavior of the compiler; they were uncovered during attempts to understand GCC's source code. It was also discovered that the caching mechanism used to speed up function size estimations provides stale data in certain cases after discovering new direct edges. This makes the compiler fail on an assertion [5, `gcc/ipa-fnsummary.cc:3480`]. We have reported this problem, but it has not been fixed yet. To overcome this limitation, we provide the `temporaries.patch` path. Its use shall become unnecessary in later releases of GCC.

The patches can be used separately as well as combined. The order of application does not matter. Either can be applied as (assuming GCC is cloned in process working directory)

```
git reset --hard cbed07845708f01a122e02016660a9152e5c14ff
git clean -dfx
git am /path/to/patch
```

A.2 Tramp3Dv4 perf

Tramp3Dv4 is analyzed in Section 3.2 with *perf*, its output is however too large to be included in the document's body. The files `perf-tramp-5.txt` and `perf-tramp-10.txt` provide the output of `perf report --stdio` after running Tramp as described in its section. The suffix number corresponds to the value of `inline-unit-growth` used during compilation.