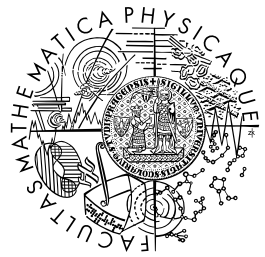

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Kubánek

Improving loop optimization with histogram profiling

Department of Applied Mathematics

Supervisor of the bachelor thesis: Jan Hubička
Study programme: Bachelor of Computer Science
Study branch: General Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

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 my advisor Jan Hubička for his technical help and enthusiasm,
and my family and friends for their patience and support.

Title: Improving loop optimization with histogram profiling

Author: Ondřej Kubánek

Department: Department of Applied Mathematics

Supervisor: Jan Hubička, Department of Applied Mathematics

Abstract: Production compilers use numerous techniques to generate performant code. One such technique is Profile-guided optimization (PGO). The principle of this technique is to insert instrumentation during compilation, gather information about program behaviour with training runs and use this information during recompilation to improve optimization.

The thesis aims to improve the precision of Loop optimizations in GNU Compiler Collection (GCC) with PGO. Currently in GCC, only the average iteration count of a loop is known with PGO. This leads to inefficiencies in both the performance and size of the binary.

We implement infrastructure for measuring more information about loop iterations and add new counters namely the histogram of iterations and histogram of iterations modulo its size. With the histogram of iterations, we improve loop peeling and implement a new case of loop versioning optimization. This significantly improves the performance of the generated code with reasonable overhead.

Keywords: GCC, compiler, loop optimization, profiling

Contents

Introduction

Profile-guided optimization (PGO) consists of three phases. During the first compilation, the compiler inserts instrumentation. Second, the user trains the generated binary on a representable inputs, which allows the instrumentation to measure interesting properties of the code and save them to the disk. Last, the code is recompiled and the saved data is loaded into the compiler, which influences the decisions of individual optimization passes. One particularly interesting type of optimization is Loop optimization. Since the behaviour of loops is heavily dependent on the runtime values, PGO offers an alluring option to assess their behaviour.

In production compilers, PGO accounts for up to 15% of additional performance on SPEC CPU 2017 benchmarks compiled with `-O2` [1]. Currently, the most common approach is to measure the *Edge profile* [2] to approximate the behaviour of a loop.

This approach is not perfect since the only information we can get about iteration counts of a loop is their average. The average count of iterations of a loop does not offer any information on the distribution of iterations of the loop. This leads to suboptimal optimizations, e.g. in cases where the average and the most common case diverge. This can lead to both performance and binary size issues since the compiler can either choose way too aggressive optimizations that worsen the size of the binary or the compiler can give up and abandon some performance.

This thesis approaches this problem by establishing infrastructure to measure other properties of loops. We also implement a histogram of iterations and a histogram of iterations modulo its size. We then use the histogram of iterations to improve the behaviour of the loop peeling optimization and introduce a new case of the loop versioning optimization.

This thesis is organized as follows. In the first chapter, we will introduce the related terms. In the second chapter, we will describe concrete details of our implementation and in the last chapter, we will show our results and discuss future work.

Chapter 1

Background

1.1 Preliminaries

Our compiler of interest is GNU Compiler Collection (GCC)[1]. In particular, we are interested in Loop optimizations and profiling.

We are also interested in interprocedural optimization, since some loop optimizations happen during it. However, as this procedure does not broadly change our approach we will omit further details [3]. For the entire thesis, we will be working with the *GIMPLE intermediate language* which is the high-level intermediate language used in GCC. Further in [4].

We briefly review standard terminology used in this thesis.

Basic block is a sequence of statements with a beginning and an end whose each execution must enter through its beginning and leave through its end. That means there are no jumps from or into the middle. In this thesis, the statements are the GIMPLE statements.

Each function is represented as a *Control flow graph* (CFG). A CFG is a directed graph, its vertices are basic blocks and its edges represent the control flow (such as jumps...) between the end of one basic block and the beginning of another one. Each CFG also has its own Entry and Exit block.

Most static compilers including GCC use intermediate languages in the *SSA-form*. SSA-form is a property of the intermediate language which demands that every variable has a single definition [5, 6]. This property can be established by simple renaming for every CFG which consists only of basic blocks with at most one incoming edge. The problem comes when two edges into the same basic block have different definitions of a variable. *Φ-nodes* are needed in this case. Φ-node is a special statement that can only be at the beginning of a basic block, its parameters correspond to incoming edges, and it defines a variable whose value

[1] `https://gcc.gnu.org/`

during execution is equal to the parameter of the Φ-node corresponding to the edge taken by the execution. In other words, Φ-nodes merge runtime definitions into a single variable. As shown in Figure 1.1. See e.g. [5, 6] for details.

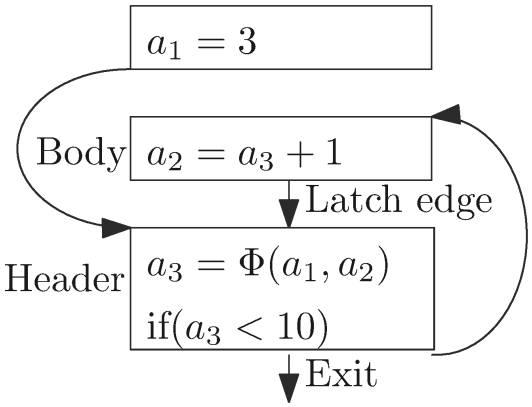

Figure 1.1 CFG showing a while loop for the variable a

Another important definition is the Dominance relation. We say that the basic block A *dominates* the basic block B if every path from Entry to B contains A. The dominance relation can be computed fast [7, 8].

1.2 Profile-guided optimization

Profile-guided optimization (PGO) consists of three phases.

Instrumentation First, the compiler inserts instrumentation into the binary to measure data. In GCC, we compile with the flag:

```
-fprofile-generate
```

GCC will also link the program with a special runtime library libgcov that will be responsible for saving the measured data [9].

Training run Then the user trains the generated binary on representative inputs. During the execution, the inserted instrumentation measures the profile and saves it to the disk. In case of GCC, it is the file with the `gcda` extension which belongs to the binary.

Recompilation Lastly, we recompile the same project with the same optimization settings as used during the instrumentation. During the compilation, the compiler will read the saved values and perform optimizations based on the gathered data. In GCC, we use the `-fprofile-use` flag.

Profiling instrumentation can be broadly divided into 4 distinct types [9].

Function profiling measures various properties of functions. GCC does not profile the number of executions of a function since Edge profiling has better accuracy. However, GCC does implement profiling of times of the first execution of functions [10]. The first example is helpful for inlining. The second one is helpful, if the compiler sorts the functions in the binary based on the time of their first execution, it increases performance significantly.

Edge profiling measures the relative number of execution counts of each edge in a CFG of a function. This allows the compiler to significantly improve, e.g. register allocation, loop unrolling and many other optimizations [9, 2].

Value profiling measures properties of concrete variables or expressions, e.g. estimating the value of b in the a/b expression. If b equals some value val most of the time. The compiler can test if b equals the value and in that case, return a/val and otherwise a/b. This allows for further optimization. GCC does implement it for various other types of counters.

Path profiling measures properties of concrete acyclic control flow paths [11]. This allows the compiler to optimize each path separately. The drawback of this technique is that there are too many different paths so the optimization decisions and the maintenance of the profile get very complex. For this reason, GCC does not implement it.

Loop histogram profiling, discussed in this thesis, falls in between Edge profiling and Value profiling. This is because we are profiling an edge but are interested in the properties of the loop represented by the edge.

1.3 Loop optimizer

Loop optimizer is a set of independent optimization passes focused on a special kind of loop called the natural loop. We call an edge $N \to M$ a *back edge* if M dominates N. For a non-empty set of all back edges with the same target T a *natural loop* is the set of T and all basic blocks on the paths in the reverse direction from their sources not containing T. Note that all of the edges from the outside of the loop must end in T of the back edges otherwise they would not be back edges. Not all strongly connected subgraphs of CFG are natural, and thus some loops are not optimized, e.g. if a `goto` is used or the empty graph. There are four important definitions when talking about natural loops as shown in Figure 1.1 [4].

Header is the basic block that is a target of the back edges.

Latch edge is a back edge and its target is the header of the loop.

Body are all of the basic blocks in a natural loop.

Exit is any edge from a basic block inside of the loop to a basic block outside of the loop.

The loop optimizer further requires that each loop has only one latch. This does not hold in general, but natural loops can be transformed to satisfy the requirement. Also, the loop optimizer mostly performs transformations only on the inner-most loops since otherwise the code could grow fast.

1.3.1 Loop analysis

Natural loop analysis finds natural loops. The portion of the loop determined by a latch edge $N \to M$ contains N, M and blocks that are on paths against the direction of the edges from N and do not contain M. If there was another entry into the body of the loop avoiding M, N would not be dominated by M since there would be a path avoiding it. Since there can be multiple latches in a loop we take the union of their respective portions as the body of the whole loop [12].

Natural loop analysis computes the natural loops from the dominance relation and determines for each basic block which is the inner-most loop it is a part of. In GCC, this information is maintained in a persistent tree data-structure for most of the compilation.

Scalar evolution analysis (SCEV) sorts scalar variables used in a loop into those that are invariant under its iterations, induction variables and variant variables. This can help with other optimizations. An *induction variable* is a variable going through arithmetic progression as the loop iterates. The value of a *invariant variable* is the same in an execution for all the iterations of the loop. Lastly, the rest of the variables are *variant* [4].

```
int l;
for (int i=0; i<k; ++i){
    l=b[i];
    b[i]=b[l];
}
```

In this loop `k` is invariant, `i` is an induction variable and `l` is variant.

Dependence analysis determines dependencies between computations in loops. It enables several loop transformations including vectorization, loop interchange, distribution and jamming, discussed in Section 1.4.3. See [12] for a more detailed description.

1.4 Loop optimizations

Since loop optimizer is organized as a set of independent optimization passes here is a brief overview of some of them. Most of them can be found in Kennedy and Allen [13], Muchnick et al. [12], and Dvořák [14].

1.4.1 Induction variable optimizations

We will start with induction variable optimizations. The goal is to find the lowest number of induction variables so that induction variable updates and computations with them are cheap. In GCC, Induction variable canonicalization can be found in the `tree-ssa-loop-ivcanon.cc` file and the rest are combined into one pass in the `tree-ssa-loop-ivops.cc` file.

Induction variable canonicalization adds a new induction variable that starts at 0 and is incremented by 1. This can help with other optimizations on induction variables and simplify some calculations.

Induction variable elimination replaces all computations with an induction variable by computations with another one. This decreases *register pressure* and if chosen well, performance does not suffer. Since we have finitely many registers and loads are expensive, removing an intermediate variable can reduce the number of unnecessary loads and stores.

For example:

```
int j=3;
for (int i=0; i<k; ++i){
    b[i]=b[j];
    ++j;
}
```

↓

```
for (int i=0; i<k; ++i){
    b[i]=b[i+3];
}
```

Induction variable strength reduction replaces an expensive computation in the loop by a cheap computation with a new induction variable. This increases register pressure, but improves performance.

For example:

```
for (int i=0; i<k; ++i)
    b[i]=b[i*2];
```

↓

```
int j=0;
for (int i=0; i<k; ++i){
    b[i]=b[j];
    j+=2;
}
```

1.4.2 Simple Loop optimizations

The second type of optimizations GCC implements are optimizations that do not depend much on dependence analysis and do not change the structure of the loop drastically.

Loop peeling copies the body of the loop multiple times and sequentially redirects latch edges to the next copy or the loop itself. If the compiler knows that the loop mostly iterates few times, it can peel the corresponding amount. This enables further optimizations and improves speculative execution. In GCC, loop peeling is in the `tree-ssa-loop-ivcanon.cc` file.

For example:

```
for (int i=0; i<k; ++i){
    b[i]+=1;
}
```

↓

```
if (k>0){
    b[0]+=1;
    if (k>1){
        b[1]+=1;
        for (int i=2; i<k; ++i)
            b[i]+=1;
}}
```

Complete loop peeling handles the case when the compiler knows that the number of iterations of a loop is bounded by some small constant c. The compiler then can peel c times to eliminate the loop, which allows us to delete the loop thus regaining mostly sequential execution. In GCC, it is in the `tree-ssa-loop-ivcanon.cc` file.

For example:

```
for (int i=0; b[i]==3 && i<2; ++i){
    b[i]+=1;
}
```

↓

```
if (b[0]==3){
    b[0]+=1;
    if (b[1]==3)
        b[1]+=1;
}
```

Copy header copies the header of the loop before its body. This can help with further optimizations if the first loop condition behaves differently, and allows for one less jump. In GCC, it is in the `tree-loop-ssa-ch.cc` file.

For example:

```
for (int i=0; i<k; ++i){
    b[i]+=1;
}
```

↓

```
int i=0;
if (i<k)
    do {
        b[i]+=1;
        ++i;
    } while(i<k);
```

As we can see it transforms a for-loop into a do-loop. It also effectively peels the loop one time without copying its body.

SCEV constant propagation can determine whether a variable is constant after the loop execution. This is not the case for the classic constant propagation. Typically, it helps when an induction variable in a well-behaved loop is limited by some invariant expression (`i<10`). SCEV constant propagation allows the compiler to determine the value of `i` after the loop. This allows for more optimizations down the line. In GCC, it is in the `tree-ssa-loop.cc` file.

For example:

```
for (int i = 0; i < 10; i++)
 ;
return i;
```

↓

```
for (int i = 0; i < 10; i++)
 ;
return 10;
```

Prefetching instructs the processor to fetch memory locations before their usage. The use of this optimization has decreased since modern CPUs implement hardware prefetching. In GCC, it is in the `tree-ssa-loop-prefetch.cc` file.

Loop invariant motion moves iteration invariant computation outside of the loop, which increases register pressure but decreases repeated computation. In GCC, it is in the `tree-ssa-loop-im.cc` file.

For example:

```
for (int i=0; i<k; ++i){
    int p=k+1;
    b[i+p]+=12;
}
```

↓

```
int p=k+1;
for (int i=0; i<k; ++i){
        b[i+p]+=12;
}
```

Loop unrolling duplicates the body of the loop several times. If the loop iterates a constant amount of times, the compiler can eliminate all unused exit conditions inside of the body, which leads to fewer branches and better performance. If we can determine the number of iterations at runtime, we can also eliminate the exit conditions, but it requires more computations. In these two cases there may be iterations left after the unrolled loop finishes, if so the compiler inserts an *epilogue* after the unrolled loop. An epilogue is mostly the copy of the original loop, but in the vectorizer even the epilogue may benefit from vectorization. For more general loops, loop unrolling can lead to further optimization. How much GCC

unrolls is dictated by the flags `-funroll-loops` which unrolls loops and `-funroll-all-loops` also unrolls loops without an induction variable. In GCC, it is in the `loop-unroll.cc` file.

For example:

```
for (int i=0; i<k; ++i){
    b[i]+=12;
}
```

↓

```
int i=0;
for (; i<k-3; ++i){
        b[i]+=12;
        ++i;
        b[i]+=12;
        ++i;
        b[i]+=12;
        ++i;
        b[i]+=12;
}
// epilogue
for (; i<k; ++i){
    b[i]+=12;
}
```

1.4.3 Loop transformation optimizations

The last category are more complicated loop transformations.

Auto-vectorization allows us to utilize *vector units* of the processor. Vector units execute multiple instructions at once. If the body of the loop is independent from its other iterations, this optimization can transform the loop to use the vector unit. In other words multiple iterations are executed at the same time. The drawback is that sometimes the compiler must also add a *prologue* because older processors require memory alignment for their vector units and an epilogue is also sometimes needed since most vector units are incapable of executing fewer instructions than they are designed for. This is not the case for newer processors that allow *masking*, but it can be also slow. Prologue and epilogue behave like the loop, but are not necessarily its copies. It can be useful to also vectorize them or transform them into a more optimized form. Vectorization is similar to

loop unrolling with pattern matching for the vector instructions. Auto-vectorization allows for a massive speed increase, but also increases the size of the generated binary. More in [15]. In GCC, the vectorizer is located in multiple files, the main being `tree-vectorizer.cc`.

Loop unswitching works for loops whose body contains conditionals that are invariant under loop iteration. The optimization transforms the loop into a conditional with two branches where each branch has a copy of the loop modified based on the result of the conditional. The invariance is determined by SCEV. With this transformation the compiler removes repeated conditional jumps. In GCC, it is in the `tree-ssa-loop-unswitch.cc` file.

For example:

```
for (int i=0; i<j; ++i){
     if (k>0)
          b[i]+=2;
     else
          b[i]-=1;
}
```

↓

```
if (k>0)
     for (int i=0; i<j; ++i)
          b[i]+=2;
else
     for (int i=0; i<j; ++i)
          b[i]-=1;
```

Loop distribution works for loops whose body has several independent computations, e.g. writing into two different arrays. The compiler can divide the loop into multiple. Each corresponds to one independent computation. Since the loops themselves are smaller it reduces register pressure and encourages auto-vectorization. In GCC, it is in the `tree-loop-distribution.cc` file.

For example:

```
for (int i=0; i<k; ++i){
          b[i]+=12;
          a[i]+=2;
}
```

↓

```
for (int i=0; i<k; ++i)
    b[i]+=12;

for (int i=0; i<k; ++i)
    a[i]+=2;
```

Loop splitting is in general a transformation which splits the loop into multiple consecutive loops over the same index range. In the simple case, there is a conditional inside of a loop where the compiler knows that the conditional holds from the start of the loop up to some constant c and then it never holds again. The compiler can simulate it by two loops, one until c and the other one from this point on. Loop splitting can simplify control flow and reduce conditional jumps. In GCC, it is in the `tree-ssa-loop-split.cc` file.

For example:

```
for (int i=0; i<k; ++i){
    if (i<50)
        b[i]+=12;
    else
        b[i]+=2;
}
```

↓

```
int i=0;
for (; i<50 && i<k; ++i)
    b[i]+=12;

for (; i<k; ++i)
    b[i]+=2;
```

Loop versioning makes multiple versions of the loop based on the properties of the particular execution. In GCC, it is in the `gimple-loop-versioning.cc` file.

In this example, if the conditional holds, the loop is easy to vectorize.

```
for (int i=0; i<k; ++i)
    b[i*s]+=8;
```

↓

```
if (s==1)
    for (int i=0; i<k; ++i)
        b[i]+=8;
else
    for (int i=0; i<k; ++i)
        b[i*s]+=8;
```

Loop jamming works if there are two independent loops over the same iteration space. The compiler can merge their bodies. This can lead to better locality for memory access. It is the opposite of loop distribution. In GCC, it is in the `gimple-loop-jam.cc` file.

For example:

```
for (int i=0; i<k; ++i)
    b[i]+=12;

for (int i=0; i<k; ++i)
    a[i]+=b[i];
```

↓

```
for (int i=0; i<k; ++i){
        b[i]+=12;
        a[i]+=b[i];
}
```

Loop interchange exchanges the inner loop with the outer loop. This is done in order to increase memory locality. For example, it is better to traverse 2D arrays first by row than by column for memory locality. This can be fixed with loop interchange. In GCC, it is in the `gimple-loop-interchange.cc` file.

For example:

```
for (int row=0; rows<rows; ++row)
    for (int col=0; cols<col; ++col)
        b[col][row]+=2;
```

↓

```
for (int col=0; cols<col; ++col)
    for (int row=0; rows<rows; ++row)
        b[col][row]+=2;
```

Predictive commoning aims to reduce the number of redundant computations in loops. This is achieved with dependence analysis. For example, when eliminating reads, the compiler looks which parts of the statements were already computed in previous iterations and makes new variables to store the results of the reads from the corresponding iterations. This obviously increases performance but increases register pressure. In GCC, it is in the `tree-predcom.cc` file.

For example:

```
for (int i=2; i<50; ++i)
    b[i]=b[i-2]
```

↓

```
int tmp0=b[0];
int tmp1=b[1];
for (int i=2; i<50; ++i){
    int tmp2;
    tmp2=tmp0;
    b[i]=tmp0;
    tmp0=tmp1;
    tmp1=tmp2;
}
```

Graphite is an experimental loop optimization framework based on polyhedral optimization. The aim of polyhedral optimization is to change the loop so that traversal through the dependence polyhedra is the most efficient. It is off by default in GCC since it can cause worse performance. In GCC, its main file is `graphite.cc` [16].

Autoparallelization tries to execute independent computations in parallel. It is also off by default since it is tricky to pick the right kind of loop. In GCC, it is in the `tree-parloops.cc` file.

1.5 Loop optimizations benefiting from loop histogram profiling

There are numerous optimizations in which iteration histogram can be used. We will delve into more detail for the most important ones. Note that prefetching, loop jamming, loop distribution, and loop interchange are also good candidates for using loop histogram since their performance is determined by whether the computation fits into the cache.

Loop peeling consists of copying the body of the loop multiple times in front of the loop itself. It is hard to determine the right amount of peeling since iteration counts of loops are heavily dependent on runtime values. In GCC, the amount is either determined by a static guess or the edge profile. The static guess is either an upper bound on iterations or that most loops iterate a lot, which is the default guess. As for edge profiling, we can get the average iteration count from the count of the latch edge divided by the sum of the counts of the exits. Therefore, we try to peel the average iteration count. This works on some types of loops and does not on others. Here is a brief overview of some types of loops.

- If the loop only iterates a lot, we do not want to peel it either way.
- If the loop mostly iterates a certain amount of times, the average will correspond to the amount so the edge count peels the right amount of iterations.
- If the loop has a normal distribution of iterations, the average would only peel half of them, but if we peeled just few more, we could peel most of the iterations.
- If the loop mostly iterates few times, but sometimes a lot, the average count will be skewed to the higher side so GCC would peel either way too many or not at all since the peeling would exceed the instruction limit.

By adding a histogram of iterations and using it for peeling and versioning, the compiler can behave better in all of the cases.

Copy header copies the header of the loop in front of the loop so that execution can be sequential during the first iteration. In the future, we can adjust the heuristic for copy header since we know how often does the loop exit without iterating. However, since this optimization is used on most of the loops, it is mildly pointless.

Loop versioning If a loop iterates mostly a certain amount of times and the iteration amount is precisely determined by an invariant variable, we can read it from the iteration histogram. This allows us to introduce a new case for loop versioning in which we test if the variable equals the probable iteration count. If so, use a completely peeled version of the loop, and if not, use the copy of the loop.

Listing 1 Usage of the optimization on a loop that mostly iterates 2 times.

```
for (int i=0; i<k; ++i)
    b[i]+=8;
```

↓

```
if (k==2){
    b[0]+=8;
    b[1]+=8;
}
else
    for (int i=0; i<k; ++i)
        b[i]+=8;
```

In the future, we could also measure different loop properties, such as the rate of entrance of the vectorizable loop in Section 1.4.3.

Auto-vectorization improves the performance of loops with larger iteration counts, however for small iteration counts it may decrease performance. For this reason, GCC implements heuristics determining the lowest number of iterations necessary to enter the vectorized loop.

Other concrete considerations are the vector size of the processor, e.g. the ARM architecture, does not guarantee the size of their vectors. In GCC, the cost model determines whether vectorization is worth it or not. Here are some of the information that determine the cost of vectorization.

- Is a prologue necessary, and if so, how large?
- Is a epilogue necessary, and if so, how large?
- Which optimizations should be used on prologue and epilogue? (e.g. peeling, vectorization, loop versioning)
- How large is the vector size if the compiler knows it?
- How fast are vector operations on the CPU?
- Does the CPU support masking?
- Can the CPU only read continuous blocks of memory, or does it have scatter and gather operations?
- Does it iterate a lot? (average iterations are high)

From the modulo histogram, we can get the number of peelings for the epilogue. From the iteration histogram, we can estimate whether it is worth it to vectorize the loop and what vector size should we use since low iteration counts with big vector sizes result in the vectorized body being unused. In addition, if the prologue or epilogue is vectorized we know that their iteration count is probably low so they are good targets for peeling.

Loop splitting If we have a loop suitable for splitting, we can add histogram counters of whether the execution stayed in one index range. If the majority of runs of the loop did, we do not want to split the loop since the branch predictor of the CPU can predict it more efficiently, and we save some space.

Chapter 2

Implementation

In this chapter, we will describe and discuss the design decisions that we used. Some bugs in GCC have been fixed as a result of this thesis. Mostly concerning the copy header optimization which did not maintain information about loops and vectorizer[1][2].

2.1 Instrumentation

We need to build infrastructure to insert the instrumentation as in Section 1.2. This starts by identifying natural loops that are suitable for instrumentation. First, we check if the loop has a real exit. There are also *fake edges* in the CFG which represent e.g. function calls. Since loops mostly return from function calls, counting them would lead to an inaccurate histogram. We also check whether we know the number of iterations of the loop. If so, there is no need to instrument the loop since the histogram would not contain more information than we already have.

We extend the form the loop optimizer uses over the profiling part so that every loop has one latch.

Since our interest lies in the number of iterations of a loop we need to insert instrumentation on the latch edge of the loop. Then we create a new induction variable starting from zero and incremented by one on the latch edge and push the created `histogram_value` to the list of all instrumented values. This happens in the `gimple_histogram_values_to_profile` function.

After that, the profiler starts to instrument the values which in our case are loops. We first build a call to the function that will save the induction variable to the designated location from the `libgcov` runtime. In our case, it is the

[1] `https://gcc.gnu.org/PR109690`

[2] `https://gcc.gnu.org/g:cda246f8`

__gcov_histogram_profiler function Listing 2. We insert a call to it on all real exit edges of the loop.

Listing 2 Part of the__gcov_histogram_profiler function which adds new iteration count to the counter. value in this case refers to the value of the induction variable and counters is the array of the profiling counters during runtime.

```
// add to the regular histogram
if (value < lin_size) {
counters[value]++;
} else {
gcov_type_unsigned pow2 = floor_log2(value);
gcov_type_unsigned lin_pow2 = floor_log2(lin_size - 1);
if (lin_size < tot_size && pow2 == lin_pow2) {
  counters[lin_size]++;
} else {
  if ((lin_pow2 - lin_size) + tot_size > pow2) {
    counters[pow2 + (lin_size - lin_pow2) - 1]++;
  } else {
    counters[tot_size - 1]++;
  }
}
}
// add to the modular histogram
counters[tot_size + (u_value \% mod_size)]++;
```

This symbolically transforms the instrumented loop like this:

```
while(!end()){
    foo();
}
```

↓

```
int i=0;
while(!end()){
    foo();
    ++i;
}
__gcov_histogram_profiler(counters, i);
```

The rest of the instrumentation is handled by the libgcov runtime that then merges the computed counters with the previous runs, which are saved on the disk. We then have to read the saved histograms in the compute_value_histograms function in the profile.cc file [9].

2.2 Counters

Our histogram is represented as a vector of counters. The vector has three sections one is the *Linear section*, another is the *Exponential section*, and the other is the *Modulo section*. Each counter in the linear portion represents precisely the runs with iteration counts equal to the index of the counter in the vector. The exponential part, on the other hand, stores all the higher iterations that do not fit in the linear section. If an iteration count does not fit in the linear portion, its binary logarithm determines its place. The modulo part then stores iteration counts modulo its length.

We chose this division since it aids us with optimizations and is easy to maintain throughout the compilation. At the moment the histogram portion sizes are GCC parameters that can be changed by the user with command line options in the format `-param=parameter-name=value`.

They are `profile-histogram-size-lin`, `profile-histogram-size-exp` and `profile-histogram-size-mod` respectively. We will use l, e and m for their values.

2.2.1 Maintaining counters

We save the loaded histogram in the structure representing the histograms loop. GCC maintains loops in a tree data-structure from the start of profiling until the end of the compilation.

Since the optimizations happen sequentially, we need to maintain a reasonable approximation of the histograms or decide to deallocate them. The transformations fall into two categories, either we decrease the iteration counts (peeling, copy header) or divide the iteration counts (unrolling, vectorization). We achieve this with the two functions `histogram_counters_minus_upper_bound` and `histogram_counters_div_upper_bound`.

Linear portion is easy to maintain. In the case of the `minus` function, we remove the iterations that decrease below zero, naturally adjust the rest of the linear portion and truncate the linear portion so all counters have real values. As for the `div` function, we take the roof of the division since even partial iterations are still iterations, and again truncate the counters without real values.

Exponential portion is more difficult to maintain. We assume that the distribution of iterations on each exponential interval is uniform. For the `minus` function we take the bottom portion of the exponential interval in powers of two (one half, quarter . . .) so that the highest iteration in the interval changes index after the decrement. We then move the corresponding portion of the index

to the new counter. We do not move iterations from the exponential part to the linear portion since it is better to have accurate information. For the `div` function, we take the middle of the interval and put all of the iterations to the index corresponding to the division of the half point by the divisor. We again do not try to fill the end of the linear portion with the corresponding exponential part. In general, we do not change the count of the last exponential counter because it typically contains huge iteration counts. These solutions are simple rather than optimal.

Modulo histogram is tricky. For the `minus` function we just rotate the histogram by the difference, which trivially corresponds to modulo arithmetic. As for the `div` function, it is possible to recover the histogram fully if the divisor is coprime with the size of the histogram because we have unique multiplicative inverses in the modulo arithmetic. Otherwise, powers of two which is the typical case we can recover the histogram but for a lesser modulo size.

Most of the changes are straightforward with these functions except for copy header optimization and auto-vectorization. The copy header optimizations can happen multiple times since the header of the loop changes, e.g. if we have a logical conjunction, the compiler must decide how many parts of the conjunction it wants to copy since they become sequentially the new headers. We then decide if it is probable that the first run exited through one of the copied headers. If so, we perform the same maintenance as if peeling one time, otherwise, we do nothing. We do not try to maintain the loop in the vectorizer as it is complicated.

If interprocedural optimization is enabled, we have to stream the counters into it, this is handled in the `lto-streamer-in.cc` and `lto-streamer-out.cc` files.

2.3 Optimizations

In this section, we will describe our optimizations improving upon standard techniques. In particular, we will describe our approach to peeling and preserving the histogram through optimizations.

2.3.1 Loop peeling

Currently, GCC uses the average amount of iterations of a loop to determine the number of times to peel. This is insufficient since if a loop mostly iterates little and once in a while a lot, the average will be high even though the compiler should peel little. There are three main parameters that GCC uses to determine whether to peel or not. It uses `max-peel-insns` which limits the amount of instructions

generated by peeling, `max-peel-times` which limits how many iterations can be peeled (16 is the default) and `max-peel-branches` which is the maximum amount of branches on the path through the peeled sequence.

We have implemented the linear portion of the histogram to tackle this problem. Since its size is 16 by default, we lose relevant information only when there are optimizations prior to peeling. Our approach is controlled by a parameter `profile-histogram-peel-prcnt` further p used to denote the percentage of the loop iterations that must be peeled away for each copy of the loop body. We consider peeling k times if the partial sum of iterations since the last peeling candidate i is at least $(k - i) * p$ percent of the total iteration count.

This prevents us from peeling iterations succeeding an iteration with a high percentage. We then choose the biggest peeling that does not exceed the GCC instruction limits and iteration limits. We further require that the estimated peeled iterations to instruction ratio of our peeling is better than if it took us the whole instruction limit to peel the whole linear portion for $p * l$ percent of iterations peeled. This ensures that giant loops are not peeled 1 time just because their body is lesser than the instruction limit and it has p iterations.

We also do not peel if we do not peel at least the percentage of the total iterations set by the parameter: `profile-histogram-peel-overall-prcnt`

After this, we also adjust the probabilities of the edges of the peeled copies since we know them from our histogram.

2.3.2 Loop versioning

The new optimization pass that this thesis presents is histogram-powered loop versioning. Let us say that we have a loop with one exit, and its iteration count is determined by a variable t. If the loop also iterates mostly a certain amount of times k, we can perform our optimization. We require the loop to have only one exit so we will not have exits after peeling. We make a conditional with two copies of the loop. The compiler sets the condition to test the equality of t and k. If so, we completely peel the loop k times. We know that the loop cannot exit any other way and that every run does exactly this. This reduces conditional jumps and improves optimizations downstream. The other branch contains the regular loop. As shown in Listing 1.

We do this if a single iteration in the linear portion of the iteration histogram has at least `loop-versioning-histogram-prcnt` percent of the total iterations.

2.4 New command-line options

We have also added new command-line options to control the peeling:

-fprofile-loops combined with -fprofile-generate or -fprofile-use instructs the compiler to profile loop histograms. -fno-profile-loops must be passed to both instrumentation stage (-fprofile-generate) and recompilation (-fprofile-use) to not profile loop histograms.

-fuse-histograms-in-peeling combined with -fprofile-use instructs the compiler to use loop iteration histograms to determine the optimal number of copies in the loop peeling pass.

-fversion-loops-using-histograms combined with -fprofile-use instructs the compiler to use loop iteration histogram to perform the new case of the loop versioning optimization.

-fpeel-loops-without-histogram combined with -fprofile-use instructs the compiler to peel loops using their edge profile if they do not have a histogram.

With profile feedback these flags are on by default but they can be turned off by using `-fno-` instead of `-f`.

Chapter 3

Results

3.1 Benchmarks

We will measure micro-benchmarks and the SPEC CPU 2017 benchmarks. We will further analyze overhead of our technique on the SPEC CPU 2017 benchmarks.

In our benchmarks, we disable vectorization and unrolling since the logic updating histograms after these transformations is not fully implemented.

3.1.1 SPEC CPU 2017

SPEC CPU 2017 are industry standard benchmarks published by: Standard Performance Evaluation Corporation (SPEC), which is a non-profit corporation that publishes a variety of industry-standard benchmarks to evaluate performance and other characteristics of computer systems. Its latest suite of CPU-intensive workloads, SPEC CPU 2017, is often used to compare compilers and how well they optimize code with different settings because the included benchmarks are well-known and represent a wide variety of computation-heavy programs [1].

SPEC specifies a base runtime for each benchmark and defines a rate as the ratio of the base runtime and the median measured runtime (this rate is a separate concept from the rate metrics). The overall suite score is then calculated as the geometric mean of these ratios. The bigger the rate or score, the better it is [1].

We run the benchmark on 8-core AMD Ryzen 7 5800X in single thread and report median of 3 iterations. Since we are interested in comparing code quality, we have not met all requirements for reportable runs according to the SPEC CPU 2017 standards.

The flags used for the benchmarks were

- The flags common between both of the compilations were:

```
-Ofast -march=native -mtune=native
```

- The extra flag in the instrumentation was: `-fprofile-generate`
- And the extra flags for the recompilation were:

```
-fprofile-use -fno-tree-loop-vectorize \
-fno-unroll-loops
```

We have measured the SPEC CPU 2017 benchmarks, the improvement in the geometric mean was 1% over normal peeling. The measurement files are in the attachment.

Since GCC is a production compiler, huge upswings in performance are unrealistic as it is already heavily optimized. In [1] we can see that progress from GCC 7.5 to 11.2 improved the SPEC CPU 2017 benchmarks by about 10%, which took four years of development. Main contribution to this improvement was the hardware support of AMD EPYC 7543P processor, which happened between GCC 7 and GCC 11, especially with respect to the vector code generation".

Benchmark	No Peeling	Standard Peeling	Histogram Peeling	Difference
500.perlbench_r	8.65	8.91	9.07	1.8%
502.gcc_r	10.6	10.6	10.6	0.0%
505.mcf_r	8.43	8.37	8.38	0.1%
520.omnetpp_r	6.16	6.04	6.51	7.8%
523.xalancbmk_r	6.49	6.58	6.66	1.2%
525.x264_r	8.00	7.97	8.21	3.0%
531.deepsjeng_r	6.61	6.54	6.46	-1.2%
541.leela_r	5.82	5.78	5.81	0.5%
548.exchange2_r	24.7	24.8	24.1	-2.8%
557.xz_r	5.34	5.58	5.55	-0.5%
Geometric mean	8.13	8.16	8.24	1%

Table 3.1 The table shows the SPEC CPU 2017 benchmark rates for our peeling optimizations and the difference in peeling between the Standard Peeling and Histogram Peeling

Overhead is important when evaluating whether an optimization is reasonable. We measured the change of size of the optimized binaries and of the `.gcda` files. We also measured the difference in the performance of instrumented binary with and without our technique. The setup used for measuring has not changed from the performance measurements.

The average size of a PGO-optimized binary currently increases by 1.2% when using standard peeling, and when using our technique, it only increases by 0.7%. The relative decrease of our technique in comparison to the standard technique

is 0.5%. This means that our technique not only improves the performance of the generated code but also modestly the size of the optimized binaries.

The number of counters increased by 50% on average. This also includes the 32 counters for the modulo histogram that is currently not used for optimization. So to achieve our performance increase, we only need a third of the counters since the rest of the histogram has 17 counters. This can also be improved by adding smarter heuristics for choosing loops to profile, e.g. if a loop calls `printf`, optimizing it will not help much.

From Table 3.1 we know that currently, the geometric mean of rates with PGO is 8.16. We have measured that the instrumented binary without histogram has a geometric mean of 5.55, and with it, the geometric mean is 4.94. This means that the instrumented binary without our technique runs 12% faster than the binary without it. In some types of programs, worsening performance can change how the program operates, e.g. programs with a lot of user interaction, such as games, change their behaviour. For these types of programs, it might be better to turn off the usage of histograms with `-fno-profile-loops`.

3.1.2 Micro-benchmarks

We have also designed microbenchmarks which show that in some cases our optimizations can improve the performance of the generated binary by a lot.

3.1.3 Peeling

The first one takes an argument and randomly generates the field `b` with either 100 or a number between 0 and 8 and then the behaviour of the inner loop in the `inc` function is determined by the current `b[i]`. In this case, we want to peel 8 times while the average is usually 5.

```
#include <stdlib.h>
int a[100];
int b[10000];
void
inc(int n)
{
 for (int i = 0; i < n; i++)
 {
   for (int j = 0; j < b[i]; j++)
    a[j]++;
 }
}
int
main(int argc, char **argv)
{
```

```
        int n = atoi (argv[1]);
        b[0] = 100;
        for (int i = 1; i < n; i++)
          if (!(rand () \% 100))
             b[i] = 100;
          else
             b[i] = (rand () \% 5) * 2;
        for (int i = 0; i < 100000000; i++)
                inc (n);
}
```

We compile on AMD Ryzen 9 7900X 12-Core Processor and standard compilation flags for a PGO build. Using histogram, peeling performs 100% better than without it. In addition, we used flags:

```
-O2 -fno-tree-vectorize -fno-unroll-loops
```

3.1.4 Versioning

For the second micro-benchmark, we have a benchmark for the loop-versioning optimization. In this case, the `gnu::noipa` attribute forbids inlining the loop function so GCC cannot be sure that `n` is equal to three at the time of execution of the function. Thus, it cannot be completely peeled. We can see from the histogram that the loop iterates only three times and version it.

```
int a[100];
int m=3;

[[gnu::noipa]]
void loop()
{
        int n = m;
          for (int i = 0; i < n; i++)
                  a[i]+=a[i];
}
int
main()
{
        for (int j = 0; j < 1000000000; j++)
          loop ();
        return 0;
}
```

We compile on AMD Ryzen 9 7900X 12-Core Processor and use standard compilation flags for a PGO build. Histogram versioning performs 40% better. In addition, we used flags:

```
-O2 -fno-tree-vectorize -fno-unroll-loops
```

3.2 Future work

One of the original motivations for introducing loop histogram profiling was improving loop vectorizer decisions and prologue/epilogue code generation (for which we plan to use currently unused modulo information). This is not fully implemented, since vectorizer implements many strategies for loop prologue and epilogue code generation and for each of them a special treatment of histogram is needed.

During our experiments, we encountered problems with the current implementation of loop heuristics which treats badly loops with low iteration counts[1]. The core of the problem is that until now, the vectorizer never had very good knowledge about the loop iteration histograms necessary to make precise estimates about the benefits of individual transformations. Consequently, heuristics were never fine-tuned for such scenarios (some important problems are being worked on actively[2]). We plan to solve this in cooperation with the maintainers of the loop vectorizer after the initial infrastructure is contributed to the compiler.

Another not fully solved problem is maintaining loop histograms through less frequent optimizations affecting loop iteration counts (such as loop splitting) and keeping them intact until loop unrolling (which is performed late in the optimization queue once instruction selection is done). While GCC has persistent loop information, it turns out that in several important cases the loops are lost and rediscovered. We identified and fixed the most common issue in loop header copying, but other problems remained, e.g. during the RTL expansion phase.

We did not do any thorough experimentation about optimal sizes of histograms, since this depends on the planned usage of the infrastructure in the loop vectorizer.

Histograms can also be applied in heuristics controlling various extra optimization passes such as loop splitting, jamming or prefetch code generation.

It is also possible to measure additional properties of the loops, such as the strides of arrays walked, to make more informed decisions in the loop versioning pass.

[1] `https://gcc.gnu.org/PR109690`
[2] `https://gcc.gnu.org/PR108410`

Conclusion

The goal of the thesis was to implement infrastructure for loop profiling and use it to improve the precision of loop optimizations with Profile-guided optimizations. We implemented a histogram of loop iterations and extended loop peeling and loop versioning heuristics to use the information. This improved performance for our microbenchmarks by 100% and 40%, respectively. The SPEC 2017 benchmarks improved by 1% while decreasing the size of the optimized binary when compared to the same optimization level. This shows that our approach improves precision of loop peeling and loop versioning. Note that the yearly rate of improvement of GCC on the SPEC benchmarks is typically around 2.5% for PGO.

The overhead of our technique was 50% more counters and 12% slower execution of the instrumented binary. Since only a third of the counters was used for optimization, there is potential for more optimization justifying the overhead.

There are several optimizations that can further benefit from this infrastructure, e.g. Auto-vectorization, loop jamming and loop splitting.

We believe that the thesis has achieved its goal. We plan to contribute the infrastructure to the upstream GCC compiler. The improvement of benchmarks with reasonable instrumentation overhead indicates that it should be accepted to main-line GCC.

Bibliography

[1] Martin Jambor et al. *Advanced Optimization and New Capabilities of GCC 11*. Tech. rep. SUSE Best Practices, 2022.

[2] Thomas Ball and James R Larus. "Optimally profiling and tracing programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (1994), pp. 1319–1360.

[3] Jan Hubička. "Interprocedural optimization framework in GCC". In: *GCC Developers Summit*. Citeseer. 2007.

[4] Richard M Stallman. "GNU compiler collection internals". In: *Free Software Foundation* (2002).

[5] Mark N Wegman and F Kenneth Zadeck. "Constant propagation with conditional branches". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.2 (1991), pp. 181–210.

[6] Ron Cytron et al. "An efficient method of computing static single assignment form". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 25–35.

[7] Robert Tarjan. "Finding dominators in directed graphs". In: *SIAM Journal on Computing* 3.1 (1974), pp. 62–89.

[8] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[9] Jan Hubička. "Profile driven optimisations in GCC". In: *GCC Summit Proceedings*. Citeseer. 2005, pp. 107–124.

[10] Martin Liška. "Optimizing large applications". Master's thesis. Charles University, 2014.

[11] Thomas Ball and James R Larus. "Efficient path profiling". In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE. 1996, pp. 46–57.

[12] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[13] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[14] Zdeněk Dvořák. "A New Loop Optimizer for GCC". In: *GCC Developers Summit*. 2003, p. 43.

[15] Dorit Naishlos. "Autovectorization in GCC". In: *Proceedings of the 2004 GCC Developers Summit*. Citeseer. 2004, pp. 105–118.

[16] Sebastian Pop et al. "GRAPHITE: Polyhedral analyses and optimizations for GCC". In: *proceedings of the 2006 GCC developers summit*. Vol. 6. Citeseer. 2006, pp. 90–91.

Appendix A

Using our work

A.1 Reproduction

To run the thesis, a Linux machine with a working C++ compiler is needed.

To access our work execute the commands in a suitable directory:

```
> git clone git://gcc.gnu.org/git/gcc.git git
> cd git
> git config --add remote.origin.fetch \
    +refs/users/kubaneko/tags/ThesisTag:refs/tags/ThesisTag
> git pull
> git checkout ThesisTag
```

To compile GCC it is best to follow the instructions in this tutorial since compiling GCC is, in general, difficult. Please use the following configuration flags for generating the `Makefile`: `-disable-multilib -enable-languages=c,c++`, since they are tested.

Once we are finished with the previous step we can use the modified GCC.

Listing 3 is a minimal example to compile

The flags to use for the instrumentation are: `-O3 -fprofile-generate -fno-tree-vectorize`

For the recompilation it is sufficient to change `-fprofile-generate` to `-fprofile-use` and add `-fopt-info` flag to dump the optimizations.

Listing 3 The amount of iterations in the inner loop is determined by the elements of the b field. Since the average amount of iteration is 3.5 the compiler would usually peel four times. But with our modification, it peels six times. The loop is also impacted by the Copy header optimization so the compiler peels one less time than it would normally.

```
int a[1000];
int b[]={1,1,1,1,6,6,6};
int
main()
{
  for (int i = 0; i < sizeof(b) / sizeof (int); i++) {
   for (int j = 0; j < b[i]; j++)
   {
            a[j]++;
   }
  }
  return 0;
}
```

Reproducing the example should be similar to:

```
> ./gcc -O3 -fprofile-generate -fno-tree-vectorize
> ./a.out
> ./gcc -O3 -fprofile-use -fopt-info -fno-tree-vectorize \
    p_test.cc

p_test.cc:11:22: optimized:
peeled loop 2, 6 times with histogram, 100% of executions
(without histogram would try to peel
4 times, 57% of executions; header execution count 0)
```

A.2 Attachments

The histogram.patch file contains our modifications to the GCC source code; the patch was taken against the commit:

5592679df783547049efc6d73727c5ff809ec302

Additionally, the SPEC CPU 2017 benchmark results for optimized and instrumented binaries are available inside separate directories, namely the `OptimizedPerformance` and `InstrumentationPerformance` directories, respectively.