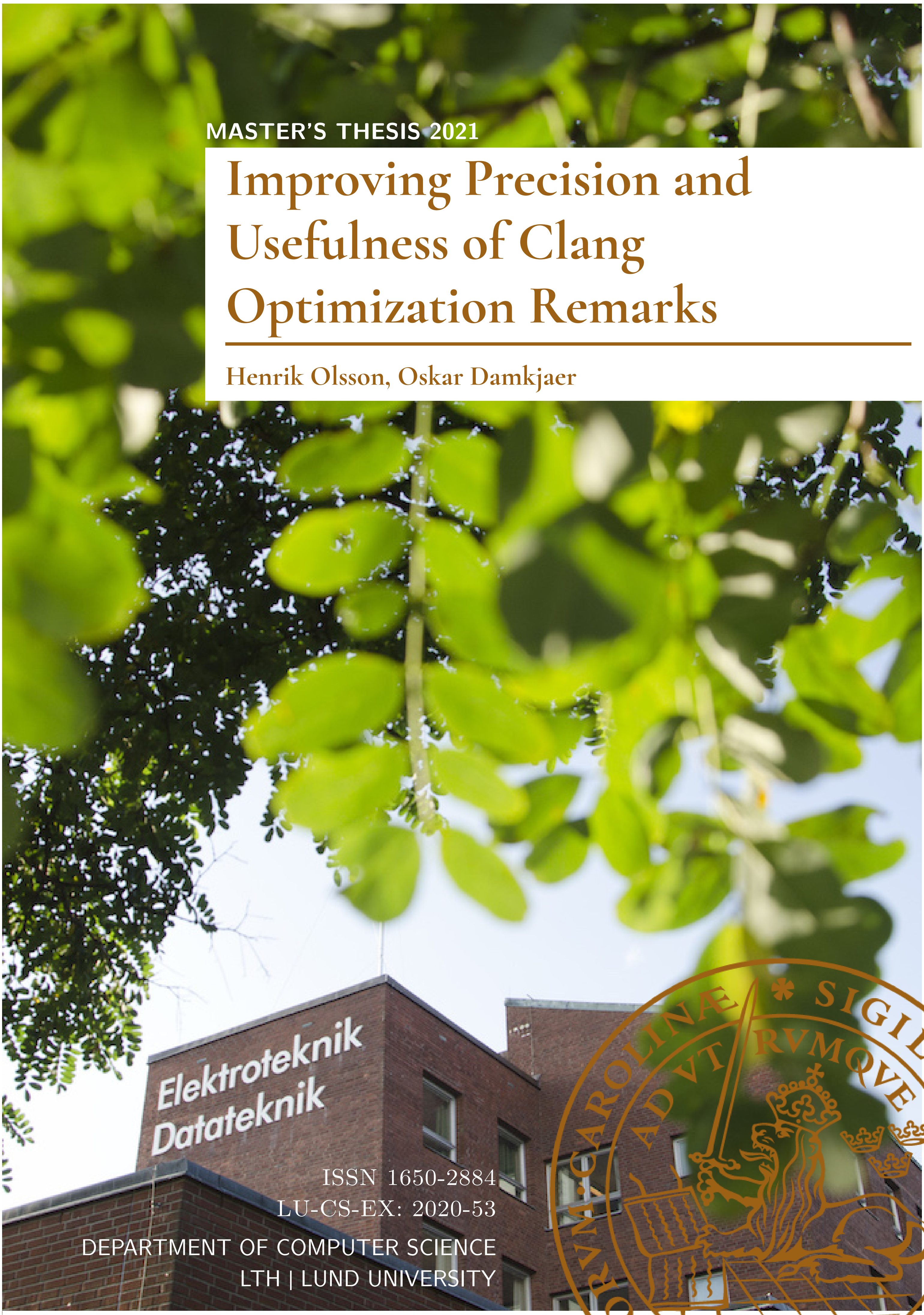
MASTER'S THESIS 2021

Improving Precision and Usefulness of Clang Optimization Remarks

Henrik Olsson, Oskar Damkjaer



ISSN 1650-2884
LU-CS-EX: 2020-53

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-53

Improving Precision and Usefulness of Clang Optimization Remarks

Henrik Olsson, Oskar Damkjaer

Improving Precision and Usefulness of Clang Optimization Remarks

Henrik Olsson
hnrklssn@gmail.com

Oskar Damkjaer
oskar.damkjaer@gmail.com

January 6, 2021

Master's thesis work carried out at the Computer Science department at Lunds Tekniska Högskola.

Supervisors: Christoph Reichenbach, christoph.reichenbach@cs.lth.se
Alexandru Dura, alexandru.dura@cs.lth.se

Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

One of the essential tools at the disposal of developers is the compiler, but despite all the work put into them, tapping into the compiler's optimizing power can be daunting and unapproachable. This thesis explores ways to make compiler optimizations in the Clang compiler, more accessible and understandable. We attempt to use debugging information to re-create names of variables and functions lost in the optimization passes to make optimization remarks more precise and actionable. We experiment with new ways to get optimization remarks from the compiler by adding pragmas to signal which functions and loops are of interest as a more precise alternative to compilation flags. We visualize optimization remarks with a Visual Studio Code extension for a faster edit, compile, check optimizations cycle, fully integrated into the editor.

Keywords: LLVM, Compilers, Error messages, Compiler aided optimization, Editor Integration

Acknowledgements

We would like to thank our supervisors Chistoph Reichenbach and Alexandru Dura for their valuable feedback, ideas and guidance.

Individual Contributions

Henrik added support for pragmas to clang and wrote the code to reconstruct the source level names. Henrik also performed the evaluations for this work and wrote the corresponding chapters. Oskar built the editor integration and set up docker and networking for the case study. He analysed the results of the case study and wrote the pieces on editor integration. Henrik made the example code and instructions for the case study. He also wrote the chapter on improvement of individual remarks and LLVM background. Oskar wrote the introduction, the related works section and the general compiler background. Both authors read, corrected and improved the report as a whole.

Contents

Chapter 1
Introduction

Compilers automatically optimize the code they compile to improve program performance. One such optimization the Clang [12] compiler supports is loop vectorization. Modern computers typically have vector instructions that can do multiple operations in parallel. Utilizing these instructions to perform multiple iterations of a loop in parallel is called loop vectorization.

Vectorizing loops can significantly increase performance but compilers can't vectorize all loops. A problem can be if the operations depend on each other within the loop. The optimization logic in compilers is complex and even their authors may be unable to predict if a loop will be vectorized.

Take for example this simple loop, can the Clang compiler vectorize it?

```
int sum[1000] = {1, 2, 3, 4};

for (int i = 1; i < 1000; i++)
{
    sum[0] += sum[i - 1];
}
```

It turns out Clang (version 10.0.0) cannot vectorize this loop with the following explanation:

"value that could not be identified as reduction is used outside the loop"

In this context a reduction variable refers to a variable that accumulates data from every iteration of a loop. It's unclear what value the message refers to, where it's used and why it couldn't be used as a reduction. Even a programmer with relevant compiler knowledge can't understand what went wrong doesn't have enough details to do so. When reading the source for loop-vectorization remarks the optimizer Clang uses, we found more debugging information. By default, optimization remarks are printed to the console where space is limited and some detail must be left out. They can also be configured to be printed in full to files. In chapter 6 we alter the optimization remarks from the loop-vectorization pass to be more flexible and include more debug data.

While improving the loop vectorization pass in an opportunity for improving the Global Value Numbering (GVN) [4] pass also arose. GVN removes redundant computations of values by re-using previous computations. A necessary precondition for doing so is that the compiler needs to know that the computation is still correct. The explanation of what code was preventing the compiler from re-using a computed value was improved to be more correct.

The previously mentioned improvements are not easily generalized. Similarly to the example of a missed loop vectorization many remarks emitted are vague as they miss the names from the source code. A reason for this is that the compiler works in several standalone modules. Clang only generates an intermediate code that a project called LLVM optimizes and generates machine code from. Because of this de-coupling, in the optimizing step of compilation the program's original representation is gone. LLVM might not even know what the source language is. The original names are not entirely lost if debug symbols are included when compiling and even though the original program structure is lost, it is possible to recreate source names. In chapter 3 we explore how well these names can be recreated and to what performance penalty.

In chapter 5, we built a plugin for the Visual Studio Code that allows programmers to get our improved optimization remarks directly in the editor. We used Clangs LibTooling to build an AST and locate function and loop headers, annotate them with an editor action. This allows us to show only the remarks affecting said loop or function and let the user pick what optimization pass they are interested in.

1.1 Research questions

RQ1: At what performance cost and to which accuracy can names of LLVM IR values be reconstructed?

RQ2: Does increased availability of optimization remarks improve programmers' ability to speed up program performance?

RQ3: Do developers perceive the feedback provided by Clang effective for enabling optimizations?

RQ4: Are scoped pragmas a good alternative to compilation flags for enabling optimization remarks?

In this thesis we present ways to tighten the feedback loop when debugging performance issues and improve the messages from the compiler. We claim our main contributions to be:

- Improve detail and correctness of error messages from two LLVM optimization passes in chapter 6.

- Reconstruct source names with an accuracy in the 60%-70% range and use them to add extra context in remark messages in 3.

- Integrate optimization remarks into the Visual Studio Code editor in chapter 5.

We also implement an alternative way to enable optimization remarks with scoped pragmas in the source code (chapter 4) instead of file-wide compilation flags to tighten the feedback loop by increasing the signal to noise ratio. This was not evaluated from a usability perspective however, due to a limited number of test users and prohibitive performance issues rendering its use infeasible in a production environment in its current form. While changes could be made to improve performance, e.g. guarding the feature behind a compiler flag, that would affect the user workflow. So user testing for the current form would not necessarily be representative of a future rearchitectured version.

Chapter 2
Background

2.1 A high level view on compilers

A traditional compiler's goal is to turn a source language into a target language, often machine code. This process is usually done in six separate steps called Scanning, Parsing, Semantic Analysis, Intermediate Code Generation, Optimization and Target Code Generation.

Scanning Turns plain text into a list of tokens representing language concepts. In this step errors such as illegal characters are detected, many languages don't allow variable names to start with numbers or contain emojis. The scanner could turn the text "while (1) sayHello() ;" into a list of tokens such as (WHILE, LPAR, INT-LITERAL, RPAR, LBRAK, FN-CALL, SEMICOLON).

Parsing Turns the token list into an abstract syntax tree (AST). An abstract syntax tree is a tree structure where language concepts are the nodes and now unneeded practicalities such as brackets can be removed. For example the tokens (WHILE, LPAR, INT-LITERAL, RPAR, LBRAK, FN-CALL, RBRAK, SEMICOLON) could become a "While" node with an "Integer literal" child and a "FunctionCall" child and the brackets and parentheses would be discarded. This step confirms that the structure of the program is valid, ie that the tree is well formed.

Semantic analysis Semantic in this context refers to the meaning of AST. It checks that the AST is a sensible program. Checks could be for example thata that the types of expressions are valid and that names have been declared before use.

Intermediate code gen Generates a simple and machine independent version of machine code from a well formed AST. The steps up until and including this one are usually known as the "front end" of the compiler since these steps are independent on the target language.

Optimization Analyzes the intermediate code to see how it can be made faster and better without changing the meaning of the program. The optimizations are known as the "middle end" of the compiler since it operates only on the intermediate code and therefore is independent on both source and target language. The optimizer often steps through the code multiple times, each time called a pass.

Target code gen Converts the simplified intermediate code into code that is executable on an actual machine. This step is called the "back end" of the compiler since it is only dependent on the target language.

We work with parts of the LLVM project, a collection of reusable compiler and toolchain technologies, of which the most relevant are Clang and the LLVM Core Libraries. LLVM previously stood for Low Level Virtual Machine but is now a name in itself and no longer an acronym.

LLVM is an open source compiler backend that has reached widespread use in recent times, mainly through its companion project Clang, which is an LLVM frontend for C, C++, Objective-C and Objective-C++. There are other compilers using LLVM as their backend however, e.g. rustc and swift. This is much thanks to the LLVM project (including Clang) being built with modularity and library components in mind, as well as a liberal license compared to GCC. LLVM being a compiler backend library does have some limitations compared to the architecture of GCC, in that it is built to be agnostic on the source language, and instead operate solely on input in its custom low-level language called LLVM Internal Representation, or IR. It does not have access to the Abstract Syntax Tree of the original source code.

2.2 LLVM IR

The IR of LLVM is a low-level language with some abstractions over assembly code to allow for more powerful analysis and greater flexibility when optimizing, and allow output to multiple different architectures. Differences to assembly languages that aid static analysis include [11]:

- Data flow representation using Single Static Assignment, where only a single code point in the entire program can assign a value to a given variable. Programs written in a language with mutable variables can be transformed into SSA form without changing the computational result [4]. This representation allows some static analyses used for compiler optimizations to be performed asymptotically faster than their non-SSA counterparts; perhaps most notably the ubiquitous GVN optimization (described in more detail in 2.5).

- A type system that can safely encode most higher-level language types while still allowing for escapes through type casts were the type system would otherwise be limiting.

- A control flow graph where the nodes (so called basic blocks) may contain multiple instructions, but the control flow may only enter at the first instruction and leave at the final one.

Another difference to assembly languages allowing for more flexibility is the effectively infinite registers, letting the issue of register allocation be deferred to a later stage. Instead of explicit registers, instructions in IR directly use (SSA) Values as operands, and are themselves Values. The equivalent of a register in assembly is thus just the name (or address, depending on if one is referring to the textual or in-memory representation) of a Value. They differ from registers however in the sense that they do not represent storage that can be read from and written to. Instead, they only name or point to the result of some computation.

An IR compilation unit is called a Module. A Module contains Functions, GlobalVariables and Metadata, which we will get to later. Functions consist of BasicBlocks, consisting of Instructions and ending with a terminal instruction, such as BranchInst. Functions, GlobalVariables, BasicBlocks, and Instructions all inherit from the Value class.

Listing 2.1: LLVM module structure [15]

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {   ; i32()*
; Convert [13 x i8]* to i8*...
% cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0

; Call puts function to write out the string to stdout.
call i32 @puts(i8* %cast210)
ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

2.2.1 Instructions

Just like in an assembly language there are many types of instructions in LLVM, e.g. for performing arithmetic. We will not describe those in detail as they are not specifically relevant for the contents of this thesis, and are usually quite intuitive in their behavior to anyone familiar with assembly languages. There are however a few instructions that are especially relevant to this thesis:

getelementptr This instruction, often shortened to GEP, takes a pointer value as its main operand, a variable number of integer offsets (whose values do not have to be statically known), and returns a pointer that has been offset relative to the main operand. If the value is a struct type the offset is the field index, while for pointers or arrays it is the array index. Since the main operand is always a pointer, the first offset will always be an array index.

load/store These take a pointer operand and a value of the same type as the pointer base type, and read or write to the memory address, respectively.

bitcast The bitcast has only one operand and returns the same value, but alters the type and as such what the bits represent. There are some restrictions on what types a value can be cast to. For example they have to be of the same size, but a pointer can be cast to any other pointer, even if the value pointed to has a different size.

phi The phi (or sometimes φ) is a special instruction in the abstracted machine model introduced for the sake of achieving SSA form, to model patterns where a value depends on previous control flow [4]. In the original source code this can be due to e.g. reassigning to a variable in one or both branches of an if statement. Since SSA does not permit such reassignment a new variable is instead created in the basic block joining the two branches, which is then assigned a value depending on which basic block the execution arrives through. A special case of this is the loop structure, where a phi assigns an iteration variable's initial value when arriving from outside the loop, and the next value when arriving from within the loop.

In LLVM, if a basic block contains any phi instructions they are always at the start of the instruction list, before any other instructions. They have the same number of operands as there are predecessors to the basic block.

2.2.2 Metadata

Metadata can be used to add contextual information to the IR. Optimization passes can use this information to affect their decision making, however it is important that the semantics of the program does not depend on the presence of the metadata [15]. A metadata node can contain module level information, like which compiler produced the IR and what the source language is, but it can also be attached to functions and other Values. The type of metadata we are interested in is debug information. This is used to embed DWARF [3] debug symbols in the output binary for consumption by debuggers such as GDB and LLDB. The debug information, when included (using the `-g` flag in the case of Clang), contains information about the types used in the source language as well as source names of variables. This will be used in Chapter 3 to reconstruct the source code of values in the IR.

Debug metadata consists of subclasses of the DINode class. Most relevant to this thesis are the subclasses of the DIType class, describing the type hierarchy of the source code; DISubprogram, describing functions or methods; and DILocalVariable, describing variables.

DIType

There are four concrete subclasses of DIType [15]:

DIBasicType Describes types such as numbers, boolean, and plain memory addresses. Includes the name of the type, their size and alignment.

DICompositeType Describes types consisting of multiple types or values, such as structs, arrays or enumerations.

DIDerivedType Describes types derived from an existing type. This can be e.g. const or atomic types. However, the relevant derived types for this thesis are pointers and struct members. Struct member types describe the fields of struct types, with the name of the struct field, the offset from the base of the struct, as well as pointing to the type of the value at the field. Relevant information stored in pointer types is the type of the value pointed to. Void pointers are represented by the base type field containing the value `null`.

DISubroutineType Describes the type of a function or method. Contains the return type (`null` if no return value) and the type of each parameter.

DILocalVariable and LLVM debug intrinsics

DILocalVariable represent a local variable in the source code (as opposed to global variables); intermediate calculations that are not stored in a variable have no DILocalVariable node. These metadata nodes are not attached to IR values directly, but are instead referenced as the second parameter in virtual function calls to special LLVM intrinsic functions (mainly `llvm.dbg.value`), whose only purpose is to link values (first parameter) and DILocalVariables together. These intrinsics do not result in any actual function calls in the final binary. A DILocalVariable instance can be referenced by multiple calls to `llvm.dbg.value`.

As the control flow reaches a call to the debug intrinsic, the referenced value now represents the local variable, until the control flow reaches a new debug intrinsic referencing the same variable. This is useful since a variable may be broken up into many values during the SSA form transformation. Debug intrinsics also contain a third parameter, of type DIExpression, which describes how the referenced value relates to the variable: e.g. the variable may be a struct where each field is a fragment represented by a separate value. [19]

DISubprogram

DISubprogram nodes can be attached to functions in the IR. They point to a DISubroutineType instance describing the type of the function, and a tuple of DILocalVariables representing the function parameters and local variables. [15]

2.3 Optimization remarks

The LLVM backend by default does not emit messages to the user very often. This is because the IR is a very low level language where semantically incorrect usage of a value rarely results in an error, but instead leaves the behavior undefined. Assuming that the frontend compiler has already type checked the source code, the resulting IR should not contain any type issues either, bar bugs in the compiler frontend. Through feature flags it is possible to enable messages known as optimization remarks [18]. If invoked through a compiler frontend, these are passed onwards to the frontend for display, otherwise they are displayed directly. The advantage of having the frontend display the remark is that it has access to the source code, and can use debug locations in the remark to display relevant context from the source code instead of just a line and column number. There are three types of optimization remarks that we consider in this thesis:

pass A remark emitted to signal that an optimization transformation was successfully performed.

missed A remark emitted to signal that an optimization transformation could not be performed, or was not deemed profitable.

analysis This category is described as analysis results that can "bring more information to the user regarding the generated code". However, many optimizations rely heavily on analysis results to decide whether a specific transformation is legal without changing the observable behavior.

A remark may contain a debug location for the general location in the source code it is relevant. It may also contain a list of key value pairs, the value of which may also contain a debug location. For simple string messages the message key will be simply "String", and the value a part of the message. Other values may be textual representations of parts of the code, such as a type or IR value, with keys describing the value kind, such as "type", "pointer" or "call". When the remark is constructed in LLVM a special value called setExtraArgs may be inserted into the list of key-value pairs. This signals that key-value pairs later in the list are ExtraArgs; they are not necessary to convey the core message of the remark, but may add extra context.

2.3.1 Remarks in Clang

Remarks can currently be enabled in Clang through the flags `-Rpass=regex`, `-Rpass-missed=regex` and `-Rpass-analysis=regex`, where `regex` is a regular expression of which all matching optimization pass names will emit remarks when invoked. The user can also opt to have *all* optimization remarks emitted to a file either in the YAML format, or a binary serialization format custom to the LLVM project, through the flag `-fsave-optimization-record` with the desired format as an optional argument. Key-value pairs occurring after setExtraArgs will be included in the optimization record, but not in the terminal output with the `-Rpass` flags.

2.4 Loop vectorization in LLVM

Modern computers have so-called Single Instruction, Multiple Data (SIMD) instructions, meaning that it's possible to do the same operation, for example addition, on multiple values in parallel by packing them into vectors. A SIMD instruction adding two vectors of length 4 then results in a vector with the equivalent of 4 scalar additions. LLVM has an optimization pass called loop-vectorize, that aims to speed up loops by doing them one vector at a time with SIMD instructions.

Since programs tend to spend a large percentage of their running time in loops, making loops more efficient can have a large effect on the total running time of the program. [5]

For a loop to be vectorizable there are a set of requirements that need to be fulfilled, the most important ones being:

- The instructions in the loop need to have a vector equivalent in the target architecture.
- The iteration count needs to be known before execution, i.e. if the compiler wants to use vector instructions to execute four loop iterations simultaneously, it needs to be able to ensure that the loop will not terminate before the next four loop iterations have passed, analogous to a loop unrolling transformation. Ideally, this is statically predictable but in dynamic checks can also be inserted in the code, conditionally executing either a vectorized or unvectorized version of the loop. [5]

- The loop cannot have backward loop-carried dependencies, meaning the loop cannot depend on the values from the previous iteration, since they will be executed simultaneously. If there are dependencies between loop iterations, they need depend with a stride longer than the vector size.
- the loop cannot have jumps or branches, other than the loop condition itself.

Since loop bodies can be quite complex, the capabilities of the compiler also play a significant role in whether a loop can be vectorized. LLVM can work around some of the requirements for vectorization, for example loops containing if statements can sometimes be vectorized even if they typically would contain branches; LLVM can eliminate some branches by replacing conditional evaluation with arithmetic operations.

2.5 Global Value Numbering

Global Value Numbering (GVN) [4] is a version of Common Subexpression Elimination. It is an optimization that removes redundant computations of values that can be shown to have been previously computed. This is done by giving each non-derived value a unique identifier, and then giving each derived value an identifier deterministically and uniquely derived from the operands and the operation. Values that end up with the same number then represent the same computation. If one of the values dominates the other – meaning that it always occurs before the other, no matter which code path is taken through the CFG – the dominated computation can be replaced with the result of the dominating computation. The GVN pass in LLVM is an improved version of the original algorithm and can also optimize some partial redundancies. In a partial redundancy some, but not all, code paths leading to a computation have already computed the same value.

Care must be taken with memory accesses, however, where a `load` may not give the same result as another `load`, even if the memory location is the same. This is because the memory location may have been clobbered – (potentially) written to – between two seemingly equivalent `load` operations. The compiler needs to check that all writes between potentially redundant memory accesses do not write to the same memory location, to avoid introducing bugs during optimization. If it cannot prove this, or that any function calls between the redundant memory accesses do not contain such operations, it has to be conservative and not remove the potentially redundant memory access. LLVM's missed optimization remark for removing the `load` in this case is improved upon in 6.2.1.

Chapter 3
Source names from LLVM IR

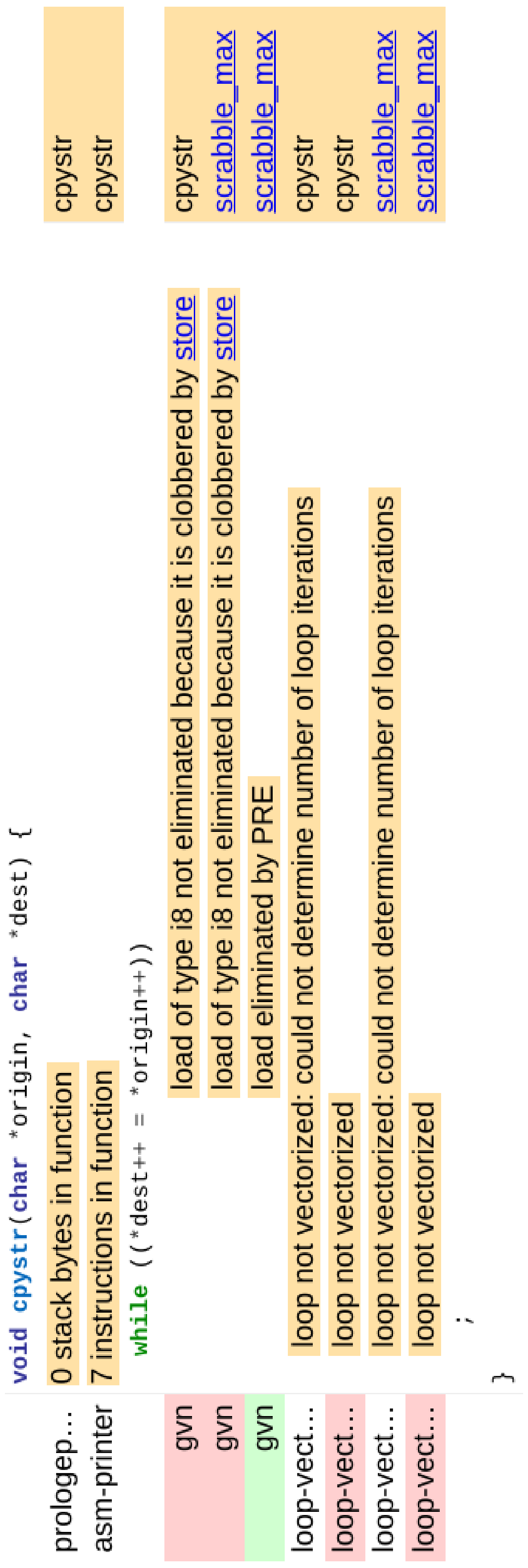
3.1 Information unavailable during optimization

Many optimization remarks are ambiguous or hard to follow due to the lack of source names as references in the remarks. The remark outputs often refer to LLVM values by the type of instruction, and their source location in the form of line and column numbers. While this may be okay for computer programs as it includes enough information to find the location of the value, this is not a very human friendly output. This is often not an issue because the remark is displayed by the compiler frontend, which can access the source code and display the relevant location for context.

However there is also the case of remarks emitted to the optimization record file: reading the YAML file directly is not a very pleasant experience since you lack all context of the source code and only see line and column numbers. Again, this is not a significant issue on its own, since it is meant to be consumed by software tools first and foremost, such as LLVM's opt-viewer and opt-diff tools [18]. As we shall see, however, tools parsing these YAML files to display in a more human readable format can offer a more streamlined user experience if the string describing the LLVM IR value contains the value as described in the source language instead of the LLVM instruction type.

As part of the LLVM project, opt-viewer [1][18] is a script that reads both optimization records and source code of a project and outputs HTML files containing the source code, with each remark visually attached to the line where it occurred, as seen in Figure 3.1. Values mentioned in the remark include a hyper-link to the line in the source code pointed at by its debug info metadata. However, the process of following hyper-links is not foolproof, as the value may just be described as a "load" in the remark, and the source line may contain multiple loads. Furthermore, even after following the link and finding out what value the remark refers to, it can be hard to remember this context coming back to the remark later since the code likely contains many other identical remarks. Therefore it would be good to

Figure 3.1: Excerpt from HTML page generated by opt-viewer



```
             void cpystr(char *origin, char *dest) {
prologep…    0 stack bytes in function                                              cpystr
asm-printer  7 instructions in function                                             cpystr
               while ((*dest++ = *origin++))
gvn                      load of type i8 not eliminated because it is clobbered by store   cpystr
gvn                      load of type i8 not eliminated because it is clobbered by store   scrabble_max
gvn                         load eliminated by PRE                                          scrabble_max
loop-vect…   loop not vectorized: could not determine number of loop iterations             cpystr
loop-vect…   loop not vectorized                                                            cpystr
loop-vect…   loop not vectorized: could not determine number of loop iterations             scrabble_max
loop-vect…   loop not vectorized                                                            scrabble_max
                 ;
             }
```

have the remark texts themselves include the original name of the value.

The reason that values are not named using their original names is that LLVM, the compiler back-end, is completely decoupled from the front-end. When LLVM starts its execution the AST may already be long gone. It may not even be known to the optimizer what the source language is, although in practice this is often included as metadata in the IR. LLVM passes operate only on LLVM IR, where value names are explicitly not guaranteed to be left unchanged [15], and are often nonsensical to a programmer not familiar with compilers. Sometimes the name of an IR value is just an integer. A real life sample we found pretty representative of optimized IR originating from Clang can be found in Listing 3.2. Full reconstruction of the source code is thus not possible in the general case. So even though the source name is available (and used by) the compiler frontend, e.g. Clang, that is only part of the full tool chain. This is why normal compilation warnings and errors can include this information, but these remarks emitted from the optimizer cannot.

3.2 Reconstructing information from debug data

If debug symbols are included in the output, present as metadata in the LLVM IR, they may enable reconstruction of variable names and other identifiers since this information is included in the final executable binary to help debuggers attach names to values and vice versa. Even with this metadata included the operation of reconstructing source names is not as trivial as it may seem, as many values in the IR are not named values in the source code, but instead derived from some chain of operations on a named value. Examples of commonly chained operations are Load and GetElementPointer.

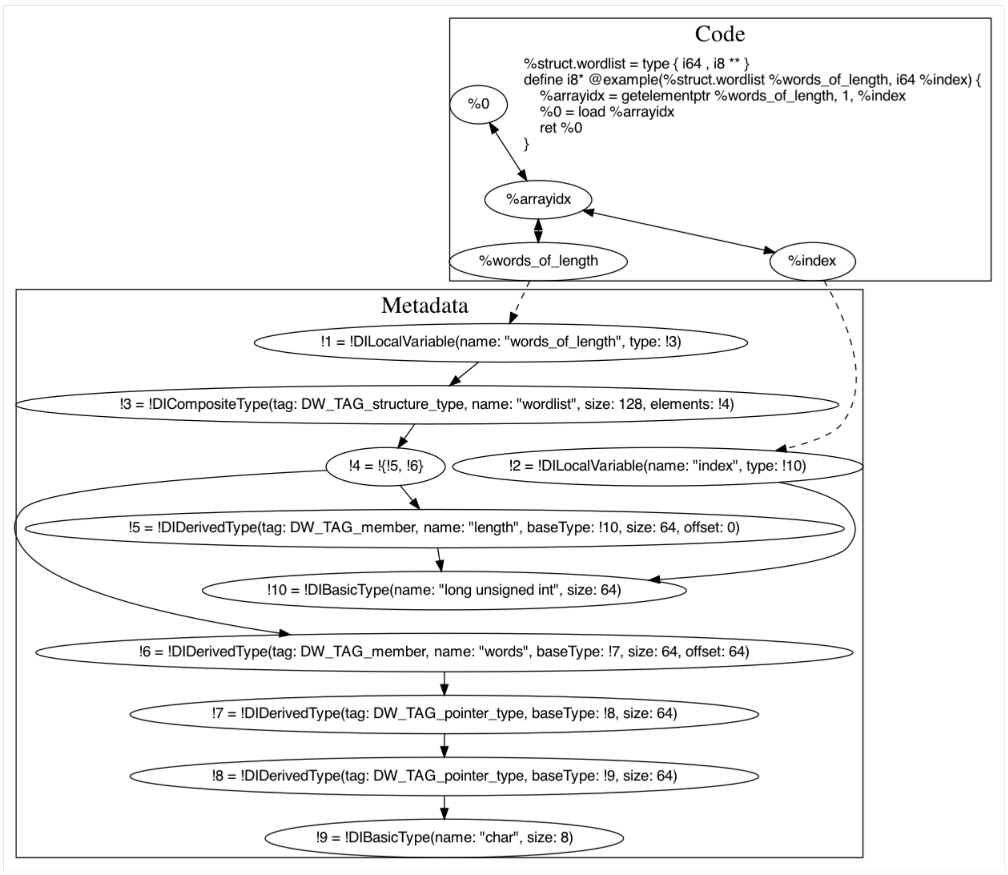
Listing 3.1: C source code for the abstracted IR and metadata graph in Figure 3.2.

```
struct wordlist {
  size_t length;
  char **words;
};

char * example(struct wordlist words_of_length, size_t index) {
  return words_of_length.words[index];
}
```

Reconstructing the name of full variables like `%words_of_length` and `%index` in the example listed in Figure 3.2 is quite trivial, by just looking at the name listed in the attached DILocalVariable metadata nodes. However, values like `%0` and `%arrayidx` have no direct metadata links, as they represent expressions consisting of multiple operations chained together that are never saved in a variable. To name such a value we need to first traverse the operands, partly to find the operand names as they are part of the full expression name, and partly to find metadata. Then we traverse the metadata debug type information to find that the second (index 1) field of the wordlist struct type is named "words" (as can be found in metadata node `!6`). Combining the names with knowledge about how they are used, `%0` can be named `*(words_of_length.words + index)`, or alternatively `words_of_length.words[index]`, which is the original name in the source code.

Figure 3.2: Illustration of metadata and simplified IR. Full arrows indicate references, dashed arrows indicate a link between IR values and metadata nodes. In real IR, these links are represented by the debug intrinsics described in 2.2.2. While normal IR values have references for uses in both directions, metadata nodes only have references in the direction of user node to used node.


```
%struct.wordlist = type { i64 , i8 ** }
define i8* @example(%struct.wordlist %words_of_length, i64 %index) {
    %arrayidx = getelementptr %words_of_length, 1, %index
    %0 = load %arrayidx
    ret %0
}
```
Code
%0
%arrayidx
%words_of_length
%index
Metadata
!1 = !DILocalVariable(name: "words_of_length", type: !3)
!3 = !DICompositeType(tag: DW_TAG_structure_type, name: "wordlist", size: 128, elements: !4)
!4 = !{!5, !6}
!2 = !DILocalVariable(name: "index", type: !10)
!5 = !DIDerivedType(tag: DW_TAG_member, name: "length", baseType: !10, size: 64, offset: 0)
!10 = !DIBasicType(name: "long unsigned int", size: 64)
!6 = !DIDerivedType(tag: DW_TAG_member, name: "words", baseType: !7, size: 64, offset: 64)
!7 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !8, size: 64)
!8 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !9, size: 64)
!9 = !DIBasicType(name: "char", size: 8)

To keep down the complexity of the code, we currently only construct names for derived values in C syntax.

3.3 Implementation

In our example program given in the task for the user evaluation (see Appendix B) the expression `words_of_length.words[j]` is represented by `%0` in Listing 3.2 after optimizations in LLVM (when compiled with function inlining disabled for the sake of this example).

Listing 3.2: IR excerpt illustrating chained operations. In this example most values retain some resemblance to their original name, but with trailing noise introduced by various LLVM transformations. This naming format can not be relied upon to be stable.

```
;--snip--
%words_of_length.sroa.4.0..sroa_idx20 = getelementptr inbounds %struct.wordlist, ↩
    %struct.wordlist* %dict, i64 %idxprom, i32 1, !dbg !82
%words_of_length.sroa.4.0.copyload = load i8**, i8*** %words_of_length.sroa.4.0..↩
    sroa_idx20, align 8, !dbg !82
call void @llvm.dbg.value(metadata i8** %words_of_length.sroa.4.0.copyload, ↩
    metadata !69, metadata !DIExpression(DW_OP_LLVM_fragment, 64, 64)), !dbg !83
;--snip--
%j.042 = phi i64 [ %inc, %if.end ], [ 0, %for.body ]
call void @llvm.dbg.value(metadata i64 %j.042, metadata !72, metadata !↩
    DIExpression()), !dbg !84
%arrayidx5 = getelementptr inbounds i8*, i8** %words_of_length.sroa.4.0.copyload,↩
     i64 %j.042, !dbg !91
%0 = load i8*, i8** arrayidx5, align 8, !dbg !91, !tbaa !92
;--snip--

; Named metadata
!7 = !DIBasicType(name: "char", size: 8, encoding: DW_ATE_signed_char)
!18 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !7, size: 64)
!54 = distinct !DICompositeType(tag: DW_TAG_structure_type, name: "wordlist", ↩
    file: !55, line: 3, size: 128, elements: !56)
!55 = !DIFile(filename: "./scrabble.h", directory: "/home/dat14hol")
!56 = !{!57, !61}
!57 = !DIDerivedType(tag: DW_TAG_member, name: "length", scope: !54, file: !55, ↩
    line: 4, baseType: !58, size: 64)
!58 = !DIDerivedType(tag: DW_TAG_typedef, name: "size_t", file: !59, line: 46, ↩
    baseType: !60)
!59 = !DIFile(filename: "/usr/local/lib/clang/10.0.0/include/stddef.h", directory↩
    : "")
!60 = !DIBasicType(name: "long unsigned int", size: 64, encoding: DW_ATE_unsigned↩
    )
!61 = !DIDerivedType(tag: DW_TAG_member, name: "words", scope: !54, file: !55, ↩
    line: 5, baseType: !62, size: 64, offset: 64)
!62 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !18, size: 64)
!69 = !DILocalVariable(name: "words_of_length", scope: !70, file: !3, line: 59, ↩
    type: !54)
!72 = !DILocalVariable(name: "j", scope: !73, file: !3, line: 60, type: !58)
```

The major algorithms used to name values can be found in Appendix C.

To name a value, we start by checking if we can name it directly, either because it is a constant of a primitive type, or because it has debug metadata attached to it. For derived values, we solve the lack of direct knowledge of the name by first naming the operands, and then constructing the name of the derived value from those and the operation itself. This process goes on recursively until we hit base case values that we can name directly.

One of the more interesting operations to name is the GetElementPtr instruction, constructing a new pointer with a certain offset to another pointer. To properly name this new derived pointer, we not only need the name of the pointer operand, but also the name describing the offset. This offset can represent one (or even several, since one GEP instruction can contain several offsets) of multiple source code concepts, such as array indexing or struct field indexing. To know what a struct field name is called, we need debug metadata describing the dereferenced data type, since LLVM IR struct types operate more like tuples and do not actually contain any field names. Through this metadata, describing a struct type with three different fields and their names, we can get the struct field name of the element with a given offset. This metadata is attached to the metadata of the ancestor value however, and not necessarily directly the value with the type itself. To access this value we need our recursive naming method to return not only the name of the parent value, but also its DIType metadata so we can traverse it in the reverse direction when returning from our method.

To illustrate this, let us walk through the example in Listing 3.2, attempting to name `%0`. Since `%0` is a `load`, we first try to name its pointer operand, `%arrayidx5`, which is a `getelementptr`. `%arrayidx5` has two operands: its base pointer, `%words_of_length.sroa.4.0.copyload`, and one offset, `%j.042`. `%j.042` is relatively straightforward to name, since it is referenced in a `llvm.dbg.value` intrinsic, attaching the `DILocalVariable` metadata `!72`, which includes the original variable name "j". Naming `%words_of_length.sroa.4.0.copyload` is slightly more complicated: there is a corresponding `DILocalVariable (!69)` naming it as "words_of_length", however the source expression we started with was "words_of_length.words[j]", not "words_of_length[j]". We can extract this information with `!DIExpression(DW_OP_LLVM_fragment, 64, 64)` included in the `llvm.dbg.value` intrinsic, and the `DICompositeType` metadata `(!54)` attached to `!69` as the type of the variable. The fragment expression metadata contains two numbers, the offset and the size of the value referenced by the intrinsic, compared to the attached metadata. We know from `!54` that the type of the variable is a struct with two elements, `!57` and `!61`. Iterating through these, we find that `!61` is a `DIDerivedType` representing a struct field with offset 64 and size 64. Once we have found the matching field, we can fetch its name, in this case "words". [1] We thus name `%words_of_length.sroa.4.0.copyload` "words_of_length.words", which we can use to name `%arrayidx5`. Now, the array indexing operator in C handles both offset and dereferencing, so we can either name it "words_of_length.words + j", or "words_of_length.words[j]". We decided to go with the latter since we found it clearer that it was a pointer offset and not just a common numerical addition, and instead added a special case to the naming of `load`s, where a dereference asterisk is not added if the pointer operand is a GEP.

Listing 3.3: Equivalent IR to Listing 3.2, but with missing llvm.dbg.value intrinsic for `%words_of_length.4.0.copyload`.

```
%struct.wordlist = type { i64, i8** }
%words_of_length.ptr = ...
call void @llvm.dbg.value(metadata %struct.wordlist* %words_of_length.ptr, ↩
    metadata !69, metadata !DIExpression()), !dbg !78
%words_of_length.sroa.4.0..sroa_idx20 = getelementptr inbounds i8***, %struct.↩
    wordlist* %words_of_length.ptr, 0, i32 1, !dbg !82
%words_of_length.sroa.4.0.copyload = load i8**, i8*** words_of_length.sroa.4.0..↩
    sroa_idx20, align 8, !dbg !82
;--snip--
%j.042 = phi i64 [ %inc, %if.end ], [ 0, %for.body ]
```

[1]This can also be applied to nested structs, for example if the fragment was instead specified as having size 32, and the matching field was itself a struct with size 64 bits, we can look at the fields inside the nested struct.

```
call void @llvm.dbg.value(metadata i64 %j.042, metadata !72, metadata !↩
    DIExpression()), !dbg !84
%arrayidx5 = getelementptr inbounds i8*, i8** %words_of_length.sroa.4.0.copyload,↩
     i64 %j.042, !dbg !91
%0 = load i8*, i8** %arrayidx5, align 8, !dbg !91, !tbaa !92
;--snip--

; Named metadata
!7 = !DIBasicType(name: "char", size: 8, encoding: DW_ATE_signed_char)
!18 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !7, size: 64)
!54 = distinct !DICompositeType(tag: DW_TAG_structure_type, name: "wordlist", ↩
    file: !55, line: 3, size: 128, elements: !56)
!55 = !DIFile(filename: "./scrabble.h", directory: "/home/dat14hol")
!56 = !{!57, !61}
!57 = !DIDerivedType(tag: DW_TAG_member, name: "length", scope: !54, file: !55, ↩
    line: 4, baseType: !58, size: 64)
!58 = !DIDerivedType(tag: DW_TAG_typedef, name: "size_t", file: !59, line: 46, ↩
    baseType: !60)
!59 = !DIFile(filename: "/usr/local/lib/clang/10.0.0/include/stddef.h", directory↩
    : "")
!60 = !DIBasicType(name: "long unsigned int", size: 64, encoding: DW_ATE_unsigned↩
    )
!61 = !DIDerivedType(tag: DW_TAG_member, name: "words", scope: !54, file: !55, ↩
    line: 5, baseType: !62, size: 64, offset: 64)
!62 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !18, size: 64)
!69 = !DILocalVariable(name: "words_of_length", scope: !70, file: !3, line: 59, ↩
    type: !54)
!72 = !DILocalVariable(name: "j", scope: !73, file: !3, line: 60, type: !58)
```

Let us revisit the naming of `%words_of_length.sroa.4.0.copyload`, but now with the IR structure in Listing 3.3. In this example, the naming of this `load` cannot be done through a direct link to metadata, as there is no `llvm.dbg.value` intrinsic linking it to metadata. Instead, we use the pointer operand to name it. This operand is a GEP (`%words_of_length.sroa.4.0..sroa_idx20`), but this time with 2 offset operands. The first offset is the array offset, which is 0 in this case since we want the actual struct that the pointer is pointing to. The second offset is the offset inside this struct, zero indexed, meaning 1 in this case selects the second field in the `%struct.wordlist` struct. The LLVM struct type carries no information of the struct field names, however. Instead we return this debug metadata from that call to name `%words_of_length.ptr`. When naming this value, no matter its type, we find that it has a `llvm.dbg.value` intrinsic linking it to metadata `!69`, this time with no offset. The naming function thus returns the name as "words_of_length", but also returns its `DIType` as `!54`. The caller, trying to name `%words_of_length.sroa.4.0..sroa_idx20`, uses this metadata to find that the second field in the struct (`!61`) is named "words", and thus returns "words_of_length.words". Again, the naming of `%words_of_length.sroa.4.0.copyload` returns the same name because of the special case that assumes GEPs already have the dereferenced name.

3.3.1 Calibrating DIType metadata

There are cases where the debug type metadata returned from recursive application of the naming method does not contain the DIType metadata needed to be returned to an outer call to the naming method. A simple example of this is once again the GetElementPtr instruction. It has a pointer operand and some offsets. We fetch the name for the pointer, together with its DIType. The DIType instance will be of the DIDerivedType subclass and represent the pointer type. From there we get the base type, which may be e.g. a struct type and represented by a DICompositeType instance, and within the DICompositeType instance we find the DIType of the field at the offsets specified in the GetElementPtr instruction. However, this DIType instance does not match the type of the GetElementPtr value: the value is a pointer to the field, while the DIType is the type of the field itself. The DIType metadata is structured as a graph with edges that can be followed in one direction only: a pointer type references its base type, but the base type has no reference to the types that reference it. There may not even be a DIType instance in the module representing this pointer type. In this case we construct a DIDerivedType to represent the pointer type ourselves, however this does require mutable access to the Module. This is why we chose to model this as a class with methods instead of top level functions, to avoid having to pass around a mutable reference to the Module (we do not always have mutable Value instances, and as such can only get an immutable reference to the Module from the Value) and the accompanying DIBuilder instance for the cases where we need it.

To help find and fix cases where we have a mismatch between the debug info type and the value type we have developed a comparison method which recursively compares a Type with a DIType using structural equality [24]. It returns either a Mismatch if differences were found between the types, Match if no significant differences were found, or IncompleteTypeMatch if they did not mismatch, but one of the Matching types was (or contained) the equivalent of a void pointer type in C: a pointer to an empty struct for Type, or a pointer whose base type is `null` for DIType. The IncompleteTypeMatch case becomes relevant later.

A less trivial example of an instruction that can result in mismatch between Value type and DIType is that of the bitcast instruction: since any pointer type can be cast to any other pointer type its impossible to completely predict how the types relate to each other, however there are some common patterns that we can look out for.

The first pattern is the upcast pattern, where a pointer to a struct type is cast to a pointer to the type of the first field of the struct. Because they have the same memory address, it is only a matter of interpretation whether a pointer points to the struct or the first field. This pattern is common in object oriented programming, where a base type is laid out as a struct in the first field of the subtype struct. To identify this case we check if the value is of a pointer type with a composite type as the base type, and is cast to another pointer type. We then recursively compare the type of the first field with the base type of the cast target type, until we either find a match or a non-composite type. If we find a match we have found an instance of the upcast pattern. The recursive comparison is necessary since the inheritance chain can be arbitrarily long. To resolve the correct DIType we instead compare the base type of the DIType instance for the bitcast operand to the target Value type. If it matches we return this DIType. If not, we repeat the same check with the first field type in the place of the base type until we find a match or a non-composite type. If we find a non-composite type that does not match the cast target type we return no DIType instance.

The second pattern is the downcast pattern, where a pointer to a type (often struct) is instead cast to a type which has this type as the first field, i.e. the inverse of the upcast. This case is trickier because not only do we not have a reference from the base type to the subtype to traverse, but we cannot just construct it ourselves, in contrast to the missing pointer type for the GEP instruction. This is because the struct fields in a DICompositeType are not the field type directly, but a DIDerivedType with the DW_TAG_member which contains the name of the field as declared in the struct. These fields are the reason we want the DIType of the struct in the first place, so we have to look for an existing instance. To do this we traverse all DITypes in the module and look for struct DICompositeTypes which reference the original DIType in their first field, and see if one matches the cast target type of the bitcast instruction. If we do not find one that matches, we try again with struct types that reference any of the last iteration's DITypes in their first field, again to support inheritance chains longer than one step. To avoid an exploding runtime due to the quadratic time complexity of searching through all DITypes once for every DIType (in the worst case) we preprocess the DITypes of the Module and cache a def-use chain in our class instance, instead making the worst case linear in time complexity.

Here the IncompleteTypeMatch becomes relevant, as we search among all the DITypes in the entire Module. In larger code bases it becomes increasingly likely that we encounter another type which matches the Value type modulo void pointers, before we encounter the actual type. When we encounter an IncompleteTypeMatch we save the matching DIType, but continue the search and return it later if we did not end up matching anything else. When we encounter a perfect match however, we return immediately.

Chapter 4

Pragmas for remark output

4.1 Emission of optimization remarks in Clang

LLVM and Clang currently have the capability to select which optimization passes should emit remarks through compilation flags. This is split up into three categories of remarks: pass, pass-missed and pass-analysis. For each category the passes for which to activate remarks can be specified using a list of regular expressions giving the programmer a lot of flexibility over which remarks to emit. However, if many or all passes are activated, the output can be large, especially in large projects. If emitted as YAML files to be fed into opt-viewer, the opt-viewer's parsing of the remarks may take minutes. Potential solutions for this performance issue has been discussed, but it has not yet been resolved. [2] A C source file can be compiled separately and then linked to the rest of the project, allowing the programmer to only activate remarks for that file through compilation flags. Depending on the build environment, this might be a relatively large operation however. On top of this, LLVM outputs a separate YAML file of remarks for each module (corresponding to a source file in C) by default, so even when outputting remarks for a full project it is relatively easy to only get the remarks for a single file. Still, for a large file, the output may be too large to be human friendly.

4.2 Pragmas as an alternative to compilation flags

To increase the flexibility of remarks, we added pragmas that allow selectively activating remarks for a given pass for a certain code region. We support three different granularities: file, function or loop level. The remarks, just like the compilation flags, also specify the remark type to activate, and a regex of which matching passes will have the pragma applied.

Listing 4.1: C program with examples of pragmas for enabling various remarks at various scopes

```c
#pragma Clang remark_analysis file(".*")
int id(int n) {
  return n;
}
#pragma Clang remark funct("licm")
int sum(int a[], int n) {
  int sum = 0;
#pragma Clang remark_missed loop("loop-vectorize")
  for (int i = 0; i < n; i++) {
    sum += a[i];
  }
  return sum;
}
```

This allows the programmer to narrow down the scope of the remarks to only the parts of the code they are interested in, increasing the signal to noise ratio. Depending on the workflow, it could also reduce the friction of obtaining remarks by allowing the programmer to never even leave their editor, if they e.g. have the compiler set to automatically recompile on save. When using a build system where all files are compiled using the same flags, it also allows the programmer to iterate quickly by not having to do a full rebuild each time the global compilation flags are changed. Instead it is enough to only recompile the changed file and its dependents.

4.3 Implementation

Since pragmas are parsed and handled in the compiler frontend while optimization remarks are constructed in the backend, a way to communicate regional activation of remarks is needed. To accomplish this we extended Clang to annotate the IR with metadata. These metadata correspond to the compilation flags for remarks, starting with a string describing whether the metadata controls passed, missed, or analysis remarks, then followed by strings (one is often enough, but we support several in the same metadata node) corresponding to the regex argument to the compiler flags. This metadata can be attached on the module level, to functions or loops, each corresponding to their respective pragmas, as shown in Listing 4.2. File scoped remarks are specified in the module level metadata `!llvm.remarks`, which in our example points to `!1`. Function scoped remarks are attached to the function as metadata labeled `!llvm.remarks`, in our example pointing to `!3`. Since loops are not a first class construct in LLVM, metadata cannot be attached directly to it. However, LLVM still has an analysis pass to reason about loops canonicalized to a certain structure [16]. For loops of this structure, there is a convention in LLVM that metadata attached to the loop latch (the conditional branch at the end of a loop iteration, determining whether to exit the loop or start a new iteration) with the label `!llvm.loop` is considered to apply to the loop as a whole. So this is where we attach the loop scoped remark metadata, in this example `!9`;

Listing 4.2: Listing 4.1 compiled with our modified version of Clang, at optimization level -Oz to minimize size.

```
; ModuleID = 'pragma.c'
source_filename = "pragma.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8↩
    :16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: minsize norecurse nounwind optsize readnone uwtable
define dso_local i32 @id(i32 returned %n) local_unnamed_addr #0 {
entry:
  ret i32 %n
}

; Function Attrs: minsize norecurse nounwind optsize readonly uwtable
define dso_local i32 @sum(i32* nocapture readonly %a, i32 %n) local_unnamed_addr ↩
    #1 !llvm.remarks !3 {
entry:
  %0 = sext i32 %n to i64
  br label %for.cond

for.cond:                                         ; preds = %for.body, %entry
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]
  %sum.0 = phi i32 [ %add, %for.body ], [ 0, %entry ]
  %cmp = icmp slt i64 %indvars.iv, %0
  br i1 %cmp, label %for.body, label %for.cond.cleanup

for.cond.cleanup:                                 ; preds = %for.cond
  ret i32 %sum.0

for.body:                                         ; preds = %for.cond
  %arrayidx = getelementptr inbounds i32, i32* %a, i64 %indvars.iv
  %1 = load i32, i32* %arrayidx, align 4, !tbaa !4
  %add = add nsw i32 %1, %sum.0
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  br label %for.cond, !llvm.loop !8
}

attributes #0 = { minsize norecurse nounwind optsize readnone uwtable "correctly-↩
    rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-↩
    pointer"="none" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-↩
    infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-↩
    signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-↩
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse↩
    ,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { minsize norecurse nounwind optsize readonly uwtable "correctly-↩
    rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "frame-↩
    pointer"="none" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-↩
    infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-↩
    signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-↩
    buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse↩
    ,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.remarks = !{!1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"remark_analysis", !".*"}
!2 = !{!"clang version 10.0.0 (git@github.com:hnrklssn/thesis-llvm.git ↩
    f1d3279e408db9f43e350e347c8945cbe695ee51)"}
!3 = !{!"remark", !"licm"}
!4 = !{!5, !5, i64 0}
!5 = !{!"int", !6, i64 0}
!6 = !{!"omnipotent char", !7, i64 0}
!7 = !{!"Simple C/C++ TBAA"}
!8 = distinct !{!8, !9}
!9 = !{!"remark_missed", !"loop-vectorize"}
```

We then extended LLVM also to search the IR for these metadata nodes, in addition to the passed compilation flags, when deciding whether to emit the remark or not. For completeness sake, we also extended Clang with a file level pragma to activate optimization record output for the current file only.

4.4 Limitations

When developing the pragmas for remark output, we initially hoped to be able to use this feature in our IDE integration to reduce parse times (as well as serialization time) by limiting the YAML dump to only include remarks originating in certain regions of the source file. However, the optimization record output is structured to be a full record of every remark, and if activated instantly outputs every constructed remark. Changing this to add a filter was deemed too invasive of a change both to the structure of the LLVM source code, as well as the semantics of the compilation flag. We still see a potential use for this type of filtered optimization record, but it would likely need to be introduced as a wider discussion, and potentially as a similar but separate compilation flag instead of changing the current behavior.

Chapter 5
Editor integration

The pragmas print remarks to stdout and therefore they don't contain all the relevant information available, Clang only emits full remarks as YAML files. As the remarks are sent in standard out they are removed from their source code context. Using opt-viewer, introduced in the previous chapter, relieves some of these issues. It works by parsing YAML files emitted from Clang and building static HTML pages of the source code with remarks included. Using opt-viewer requires the programmer to run some commands and then make a context switch to a web browser making the edit compile check cycle slower than it could be. Apart from these considerations when working with the pragmas, you need to know what optimization passes exist and read up on the syntax for using them.

5.1 Integrating remarks into an editor

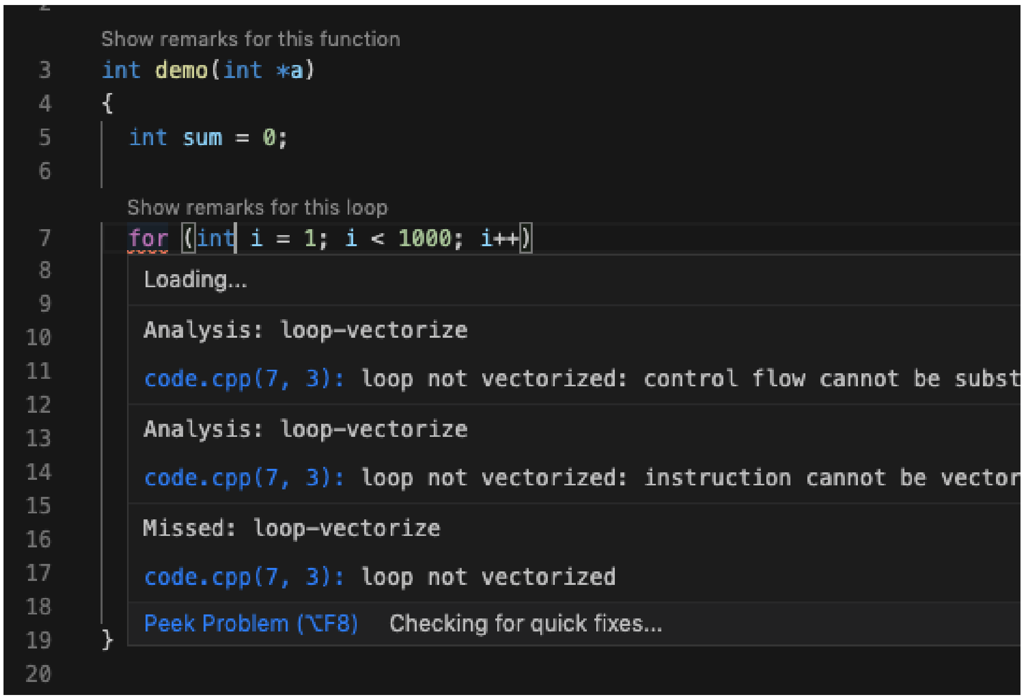
To shorten the time between editing a file and seeing if it had the intended effect we built an editor extension (figure 5.1) for the open source Visual Studio Code editor (VS Code) from Microsoft. It provides a faster way to see remarks the programmer cares about on a loop or function level. Like opt-viewer, we also include all types of remarks for the selected region, increasing the discoverability of `analysis` remarks providing context to `missed` remarks.

To function our tools needs to:

- know the flags reqiured to run the program properly
- know the location of functions and loops in the source
- compile the program to generate YAML output and show the remarks in source

To properly run a C++ program requires knowing the compile time flags. To avoid having to manually set up the flags for each file and making sure they are correct, build tools such as cmake can generate a compilation database containing this information. We provide instructions on how to generate a compilation database in our documentation. Users will hopefully

Figure 5.1: Screenshot of a remark shown in the VS Code extension.


```
Show remarks for this function
int demo(int *a)
{
  int sum = 0;

  Show remarks for this loop
  for (int i = 1; i < 1000; i++)
    Loading...
    Analysis: loop-vectorize
    code.cpp(7, 3): loop not vectorized: control flow cannot be subst
    Analysis: loop-vectorize
    code.cpp(7, 3): loop not vectorized: instruction cannot be vector
    Missed: loop-vectorize
    code.cpp(7, 3): loop not vectorized
    Peek Problem (⌥F8)   Checking for quick fixes...
}
```

already have used a compilation database to set up language support in VS Code. When we can compile the program to get output remarks, we want a way to filter only the relevant remarks. We annotate functions and loops in the editor, enabling a user to get remarks for those sections, rather than a full file. To do so we need to know where in the code they are located. Parsing C++ code is more complex than it might seem, and a best effort regex solution to find locations of function and loops would include both false positives and negatives. To solve this issue we used Clangs LibTooling[14] to build a standalone executable that can accurately locate loop and function declarations.

The VS Code extension runs as a normal Node.js process which allows extensions to spawn processes on the host machine, so we are able to use the compilation database to spawn a Clang process and collect its YAML output. We then parse the YAML, ask the user what passes interest them and filter out the remarks from other passes and those that are outside the relevant parts. These are then turned into diagnostics in the editor shown on hover

5.2 Implementation

5.2.1 Visual Studio Code

VS Code is an open source, language agnostic code editor from Microsoft. In the stack overflow developer survey of 2019 it ranked the most widely used code editor among the respondents[22].

The editor is extendible and has strong support for extensions, its language support and some core features use the same extension API as outside developers get access to. Notable features used in our extension are the so-called Code Lenses and the diagnostics. Code lenses are subtle links located above a line of code, that trigger an editor action when clicked. A common use-case for code lenses is a quick way to run or debug a function. VS Code represents diagnostics like most other editors, as squiggly lines under the code where the diagnostic is relevant and provide more details when hovered.

Before opting to build an extension for VS Code we considered using Language Server Protocol (LSP)[21] for cross-editorial support. LSP is meant to standardize how development tools and language servers communicate so that every editor doesn't have to reimplement language specific features such as auto complete. The LLVM project includes a language server for C and C++ code called Clangd [13]. Initially the idea was to build upon Clangd as a way to integrate remarks into editors. Clangd does syntax level actions and does not generate any compiler remarks, we would need to make large modifications for it to do so. We considered building an intercepting language server to combine with Clangd but decided to build a standalone VS Code extension instead as it would be less complex and easier to distribute.

5.2.2 Finding function declarations and loops in C++ code

Building C++ programs consist of three steps: preprocessing, compiling and linking. In C++ code there are directives for the preprocessor such as #include and #define.

The preprocessor expands include directives to the contents of the file they refer to and the define directives expand to macros. To share function definitions between source files the language uses so-called header files which contain function definitions that the source files can include.

Even a trivial C++ program with a single include directive of a standard header can result in more than a thousand lines of expanded C++ code that are treated as a single compilation unit by the compiler. The compiler translates the compilation unit into an object file, which is binary code with some extra metadata. After the compilation the linker joins together these object files into one executable file. The middle step of object files avoids having to recompile an entire project every time a single file changes.

We needed a way to know were loops and functions where declared and since we wanted accurate results, we dismissed using a simple regular expression. C++ has a large and complex grammar that is context-dependent, meaning that a code segment can have several different valid parses and the compiler needs the full program to correctly understand the meaning of it.

Clang supports printing the AST using the -ast-dump flag, but it proved unfeasible with non-trivial C++ programs since the AST also contains all included files meaning there are false positives and that printing the AST took too long for our use case.

To solve the problem we instead use LibTooling, a part of the infrastructure Clang provides. LibTooling is a C++ interface to Clang that is used to build stand-alone tools that use Clang to get syntactic and semantic information from a C++ file. The Clang-tools included in the Clang project, such as the code formatter Clang-format, use LibTooling.

Clang provides a powerful AST matcher with a declarative syntax and a source manager class that keeps track of from what file each node comes from. With these features our implementation is straight forward, we match all variations of loop and function declarations nodes and print the ones written in the main file.

To run our program in the VS Code extension, we bundle it with the extension and spawn a process from node to run it and collect it's standard output. Our executable looks for internal Clang headers (such as stdargs.h) relative to its own location and since bundling necessarily moves the executable, it can no longer compile any non-trivial programs properly. For this reason we also bundled the built-in headers into the extension and adjusted where our program searched for the headers.

Chapter 6

Improvement of individual remarks

6.1 Lack of detail in current optimization remarks

Since remarks as presented in the compiler front-end do not provide as much detail regarding the values involved as they do in the optimization record where they include the ExtraArgs section (as described in 2.3), it's easy to overlook providing useful ExtraArgs in remarks when developing a transformation. Both opt-viewer and our VS Code plugin make these ExtraArgs easily available to the programmer, and as such we believe the usefulness of the IDE integration would be greatly improved by improving the construction of these remarks themselves. There is also a symbiotic effect with the source name reconstruction in that the values may be even more useful when their names are provided, and the introduction of source names provides more value when more remarks use them.

6.2 Improving GVN and vectorization remarks

To improve this situation we have looked at existing remarks with a critical eye, questioning whether there may be information missing necessary to understand what happened without reading the source code of LLVM. We then looked at the source code producing the remark and what information is available without too large structural changes to the code. Since there are many different remarks and limited time, we have focused our efforts on the low-hanging fruit.

6.2.1 GVN

Listing 6.1: Example code resulting in two missed gvn remarks, only one of which suggests another access that could have been reused. The memory accesses of pointer `a` cannot be reused as the compiler does not know whether `externalFunc` writes to it or not.

```c
void externalFunc();

int f(int *a, int b) {
    *a = b;
    externalFunc();
    int c = *a;
    externalFunc();
    *a = *a + 1;
    return c;
}
```

Listing 6.2: Original gvn remarks from the snipped in Listing 6.1. The second remark does not contain an "OtherAccess" field since the load in question has multiple dominating accesses to the same pointer.

```
--- !Missed
Pass:             gvn
Name:             LoadClobbered
DebugLoc:         { File: gvn.c, Line: 6, Column: 10 }
Function:         f
Args:
- String:           'load of type '
- Type:             i32
- String:           ' not eliminated'
- String:           ' in favor of '
- OtherAccess:      store
DebugLoc:         { File: gvn.c, Line: 4, Column: 5 }
- String:           ' because it is clobbered by '
- ClobberedBy:      call
DebugLoc:         { File: gvn.c, Line: 5, Column: 2 }
...
--- !Missed
Pass:             gvn
Name:             LoadClobbered
DebugLoc:         { File: gvn.c, Line: 8, Column: 7 }
Function:         f
Args:
- String:           'load of type '
- Type:             i32
- String:           ' not eliminated'
- String:           ' because it is clobbered by '
- ClobberedBy:      call
DebugLoc:         { File: gvn.c, Line: 7, Column: 2 }
...
```

Listing 6.3: GVN remarks from snipped in Listing 6.1 with improved analysis. The second remark now also includes a value for the “OtherAccess” field.

```
--- !Missed
Pass:            gvn
Name:            LoadClobbered
DebugLoc:        { File: gvn.c, Line: 6, Column: 10 }
Function:        f
Args:
  - String:          'load of type '
  - Type:            i32
  - String:          ' not eliminated'
  - String:          ' in favor of '
  - OtherAccess:     store
    DebugLoc:        { File: gvn.c, Line: 4, Column: 5 }
  - String:          ' because it is clobbered by '
  - ClobberedBy:     call
    DebugLoc:        { File: gvn.c, Line: 5, Column: 2 }
...
--- !Missed
Pass:            gvn
Name:            LoadClobbered
DebugLoc:        { File: gvn.c, Line: 8, Column: 7 }
Function:        f
Args:
  - String:          'load of type '
  - Type:            i32
  - String:          ' not eliminated'
  - String:          ' in favor of '
  - OtherAccess:     load
    DebugLoc:        { File: gvn.c, Line: 6, Column: 10 }
  - String:          ' because it is clobbered by '
  - ClobberedBy:     call
    DebugLoc:        { File: gvn.c, Line: 7, Column: 2 }
...
```

The GVN transformation emits a missed remark when it fails to reuse a value from a memory clobber in a later load of the same memory region. This remark points out the location of the load instruction as the debug location of the remark itself and reports the type of the load. It includes two extra arguments in the form of the clobbering value, and optionally another access of the same memory that could have been reused were it not for the clobbering value. The analysis finding the otherwise reusable memory access was quite limited, however. In fact, it was called out as such in a FIXME comment in the source code. Although a complete analysis was deemed too much work since memory analysis is a complex topic, some improvement was made in that more cases are handled than previously, as can be seen in Listing 6.3. The reason we do not have accessible information for which value would have been able to be forwarded were it not for the clobbering instruction is that the analysis for eliminating the load does not try to eliminate the load using another access, but instead tries to use knowledge about the clobbering instruction for reusing its value in the downstream load. When this is deemed impossible the remark is constructed, and the OtherAccess value is only added on as a hint that there is another use of the same pointer that may have been more suitable for elimination of the load. There may not exist another access to the pointer that would have been able to be reused to eliminate the load, and as such we are satisfied with a best effort analysis that is mostly correct, especially since that was also the case previously. An example of a case that we do not consider in the remark construction analysis, but is considered in the decision making of the transformation itself, is whether there are memory fences between the previous use and the load preventing the load from being eliminated.

Listing 6.4: Previous code to find other access to the `load` `LI`. From gvn.cpp.

```
/// Try to locate the three instruction involved in a missed
/// load-elimination case that is due to an intervening store.
static void reportMayClobberedLoad(LoadInst *LI, MemDepResult ↩
   DepInfo,
                                   DominatorTree *DT,
                                   OptimizationRemarkEmitter *↩
                                      ORE) {
  using namespace ore;

  User *OtherAccess = nullptr;

  OptimizationRemarkMissed R(DEBUG_TYPE, "LoadClobbered", LI);
  R << "load of type " << NV("Type", LI->getType()) << " not ↩
      eliminated"
    << setExtraArgs();

  for (auto *U : LI->getPointerOperand()->users())
    if (U != LI && (isa<LoadInst>(U) || isa<StoreInst>(U)) &&
      DT->dominates(cast<Instruction>(U), LI)) {
      // FIXME: for now give up if there are multiple memory ↩
          accesses that
      // dominate the load.  We need further analysis to decide↩
           which one is
      // that we're forwarding from.
      if (OtherAccess)
        OtherAccess = nullptr;
      else
        OtherAccess = U;

    }

  if (OtherAccess)
    R << " in favor of " << NV("OtherAccess", OtherAccess);

  R << " because it is clobbered by " << NV("ClobberedBy", ↩
      DepInfo.getInst());

  ORE->emit(R);
}
```

Listing 6.5: Our improved version of the analysis in Listing 6.4.

```
/// Assuming To can be reached from both From and Between, does↩
    Between lie on
/// every path from From to To?
static bool liesBetween(const Instruction *From, Instruction *↩
   Between,
                        const Instruction *To, DominatorTree *↩
                           DT) {
  if (From->getParent() == Between->getParent())
```

```
    return DT->dominates(From, Between);
  SmallSet<BasicBlock *, 1> Exclusion;
  Exclusion.insert(Between->getParent());
  return !isPotentiallyReachable(From, To, &Exclusion, DT);
}

/// Try to locate the three instruction involved in a missed
/// load-elimination case that is due to an intervening store.
static void reportMayClobberedLoad(LoadInst *LI, MemDepResult ↩
   DepInfo,
                                   DominatorTree *DT,
                                   OptimizationRemarkEmitter *↩
                                       ORE) {
  using namespace ore;

  User *OtherAccess = nullptr;

  OptimizationRemarkMissed R(DEBUG_TYPE, "LoadClobbered", LI);
  R << "load of type " << NV("Type", LI->getType()) << " not ↩
      eliminated"
    << setExtraArgs();

  for (auto *U : LI->getPointerOperand()->users()) {
    if (U != LI && (isa<LoadInst>(U) || isa<StoreInst>(U)) && ↩
        cast<Instruction>(U)->getFunction() == LI->getFunction↩
        () &&
         DT->dominates(cast<Instruction>(U), LI)) {
      // Use the most immediately dominating value
      if (OtherAccess) {
        if (DT->dominates(cast<Instruction>(OtherAccess), cast<↩
            Instruction>(U)))
          OtherAccess = U;
        else
          assert(DT->dominates(cast<Instruction>(U),
                               cast<Instruction>(OtherAccess))↩
                                   ;
      } else
        OtherAccess = U;
    }
  }

  if (!OtherAccess) {
    // There is no dominating use, check if we can find a ↩
        closest non-dominating
    // use that lies between any other potentially available ↩
        use and LI.
    for (auto *U : LI->getPointerOperand()->users()) {
      if (U != LI && (isa<LoadInst>(U) || isa<StoreInst>(U)) &&
          cast<Instruction>(U)->getFunction() == LI->↩
              getFunction() &&
          isPotentiallyReachable(cast<Instruction>(U), LI, ↩
              nullptr, DT)) {
        if (OtherAccess) {
          if (liesBetween(cast<Instruction>(OtherAccess), cast<↩
              Instruction>(U),
                          LI, DT)) {
```

```
                OtherAccess = U;
              } else if (!liesBetween(cast<Instruction>(U),
                                      cast<Instruction>(OtherAccess↩
                                         ), LI, DT)) {
                // These uses are both partially available at LI ↩
                   were it not for the
                // clobber, but neither lies strictly after the ↩
                   other.
                OtherAccess = nullptr;
                break;
              } // else: keep current OtherAccess since it lies ↩
                 between U and LI
            } else {
              OtherAccess = U;
            }
          }
        }
      }

      if (OtherAccess)
        R << " in favor of " << NV("OtherAccess", OtherAccess);

      R << " because it is clobbered by " << NV("ClobberedBy", ↩
          DepInfo.getInst());

      ORE->emit(R);
    }
```

The first improvement to the code in Listing 6.4 is to add a break statement after `OtherAccess = nullptr;`, to avoid the bug of outputting an `OtherAccess` if the number of matching uses is odd, even when not intended. However, this break statement will be delegated to a second for loop analyzing non-dominating uses, since we can always reason about the order of dominating uses: if uses u_1 and u_2 both dominate load l, then either u_1 dominates u_2 or vice versa, since both code points have to be executed before reaching l. This is not necessarily the case with partially available uses, and so if we have multiple partially available uses where a strict ordering cannot be determined with respect to l we break and set `OtherAccess` to `null`;

6.2.2 Loop-Vectorize

The loop-vectorizer has many remarks for different ways vectorization can fail. These are constructed and emitted in a consistent manner using a helper function. This function limits the flexibility of the structure of these remarks in that it does not allow for extra arguments, only a textual description and (optionally) a related instruction to copy the remark's debug location from. We expanded this function to accept a lambda function for adding extra arguments where relevant, and added lambdas for extra context to some call-sites. Furthermore, many cases were emitted with identical high level descriptions, while emitting debug messages describing the low level reason. For example, 8 different call sites to the reportVectorizationFailure function all resulted in identical remarks with the description “CFGNotUnderstood: loop control flow is not understood by the vectorizer”, with the following different debug messages:

- Loop doesn't have a legal pre-header
- The loop must have a single backedge
- The loop must have an exiting block
- The exiting block is not the loop latch
- Unsupported basic block terminator
- Unsupported conditional branch
- Found a non-int non-pointer PHI
- Found an invalid PHI

While these debug messages had probably been deemed too low-level to emit to the programmer when reportVectorizationFailure was initially designed (e.g. requiring knowledge of the LLVM IR language), the extra information may still be relevant to advanced users in the more verbose optimization record. We opted to include this information as an extra argument to avoid cluttering the compiler frontend remarks with rarely needed information.

Listing 6.6: Example optimization record from loop-vectorize in YAML format

```
--- !Analysis
Pass:            loop-vectorize
Name:            CFGNotUnderstood
DebugLoc:        { File: oggenc.c, Line: 1237, Column: 2 }
Function:        main
Args:
- String:          'loop not vectorized: '
- String:          loop control flow is not understood by vectorizer
...
--- !Analysis
Pass:            loop-vectorize
Name:            LoopContainsSwitch
DebugLoc:        { File: oggenc.c, Line: 1237, Column: 2 }
Function:        main
Args:
- String:          'loop not vectorized: '
- String:          loop contains a switch statement
...
--- !Analysis
Pass:            loop-vectorize
Name:            CFGNotUnderstood
DebugLoc:        { File: oggenc.c, Line: 4527, Column: 10 }
Function:        main
Args:
- String:          'loop not vectorized: '
- String:          loop control flow is not understood by vectorizer
...
--- !Analysis
Pass:            loop-vectorize
Name:            CantVectorizeLibcall
DebugLoc:        { File: oggenc.c, Line: 4527, Column: 10 }
Function:        main
Args:
- String:          'loop not vectorized: '
- String:          call instruction cannot be vectorized
...
--- !Analysis
Pass:            loop-vectorize
```

```
Name:             NonReductionValueUsedOutsideLoop
DebugLoc:         { File: oggenc.c, Line: 45361, Column: 3 }
Function:         vorbis_bitrate_addblock
Args:
- String:           'loop not vectorized: '
- String:           value that could not be identified as reduction is used ↩
    outside the loop
...
```

Listing 6.7: 6.6 with our improvements, for example explaining what was not understood with the CFG.

```
--- !Analysis
Pass:             loop-vectorize
Name:             CFGNotUnderstood
DebugLoc:         { File: oggenc.c, Line: 1237, Column: 2 }
Function:         main
Args:
- String:           'loop not vectorized: '
- String:           loop control flow is not understood by vectorizer
- DebugMsg:         The loop must have a single exiting block
- String:           '. Exiting blocks: '
- exiting block terminator: switch
  DebugLoc:         { File: oggenc.c, Line: 1237, Column: 2 }
- exiting block terminator: br
  DebugLoc:         { File: oggenc.c, Line: 1302, Column: 25 }
...
--- !Analysis
Pass:             loop-vectorize
Name:             LoopContainsSwitch
DebugLoc:         { File: oggenc.c, Line: 1237, Column: 2 }
Function:         main
Args:
- String:           'loop not vectorized: '
- String:           loop contains a switch statement
- DebugMsg:         Loop contains a switch statement
...
--- !Analysis
Pass:             loop-vectorize
Name:             CFGNotUnderstood
DebugLoc:         { File: oggenc.c, Line: 4527, Column: 10 }
Function:         main
Args:
- String:           'loop not vectorized: '
- String:           loop control flow is not understood by vectorizer
- DebugMsg:         The exiting block is not the loop latch
...
--- !Analysis
Pass:             loop-vectorize
Name:             CantVectorizeLibcall
DebugLoc:         { File: oggenc.c, Line: 4527, Column: 10 }
Function:         main
Args:
- String:           'loop not vectorized: '
- String:           call instruction cannot be vectorized
- DebugMsg:         Found a non-intrinsic callsite
...
--- !Analysis
Pass:             loop-vectorize
Name:             NonReductionValueUsedOutsideLoop
DebugLoc:         { File: oggenc.c, Line: 45361, Column: 3 }
Function:         vorbis_bitrate_addblock
Args:
- String:           'loop not vectorized: '
- String:           value that could not be identified as reduction is used ↩
    outside the loop
- DebugMsg:         Found an unidentified PHI
...
```

As can be see when comparing Listing 6.6 and Listing 6.7, some remarks provide more context, while others are only marginally more descriptive. Some remarks, like LoopContainsSwitch, have the same information in the remark message and the debug message, leading to duplicated output. While we have only yet successfully added an extra context lambda to one call-site, we have shown that it can be done successfully, and believe there is an opportunity to continue adding similar contexts to other remarks in the loop-vectorize pass. In the examples above context could be added to show which block is the exiting block and which block is the loop latch when they differ, as well as where a non-reduction phi is used outside of the loop.

Chapter 7
Evaluation

We split the evaluation of our work into three parts: a user study to gauge the helpfulness of our editor integration, the accuracy of the reconstructed source names, and the performance impact of reconstructing source names.

7.1 Usability of the editor integration

7.1.1 Method

Students taking Jonas Skeppstedts course in Algorithm Implementation (EDAF15) at Lunds Tekniska Högskola were invited to participate in a competition after finishing the final lab of the course. The task was to speed up a deliberately unoptimized program we created that found the best word to play in a game of scrabble. The code has opportunities for both algorithmic improvements and improvements allowing the compiler to optimize further.

Since the course had few participants and it is difficult to get test subjects students who had recently taken courses in Algorithms, C programming and Optimizing compilers were also invited to participate.

The participants were divided into two groups: a group using our VS Code extension, and a control group using the existing Clang compilation flags for emitting remarks. Instructions included an introduction to optimization remarks and their designated tool. Both groups used our modified version of Clang with improved remark messages and source names. Since trials were conducted remotely due to the Covid-19 pandemic we provided each participant with a url to a VS Code (technically a fork of the open source VS Code project for remote development [9]) instance hosted by us, while accessible through the browser and otherwise behaving just like the normal desktop version of VS Code. This served the purpose of keeping environments identical, bar the extension and instructions, while removing the step of installing the custom version of Clang, VS Code and our extension on the participants' computers. It also allowed us to continuously save copies of the source code every time the participants saved in the editor, allowing us to later analyze the progression of their changes to the code. For the group using our extension we also logged each time they displayed a remark, in the hopes of seeing a connection between performance improvements and displayed remarks. The students were given up to an hour to complete the task, and afterwards asked to fill out a survey with questionsA about their experience using the remarks and their experience levels using C or C++.

```
unsigned scrabble_score(char *word) {
    unsigned sum = 0;
    while(*word) {
    sum += char_points[CHAR_INDEX(*word++)];
  }
    return sum;
}

void cpystr(char *origin, char *dest) {
  while ((*dest++ = *origin++))
    ;
}

/*
 * The index of 'dict' gives the list of playable words with ↩
    word size equal to
 * that index. The 'best' buffer should contain a copy of the ↩
    optimal word when
 * finished. In case of a tie, the shorter word wins.
 */
void scrabble_max(struct wordlist dict[], unsigned ↩
   max_word_size,
                  char best[max_word_size]) {
  for (unsigned i = 0; i <= max_word_size; i++) {
    struct wordlist words_of_length = dict[i];
    for (size_t j = 0; j < words_of_length.length; j++) {
      unsigned score = scrabble_score(words_of_length.words[j])↩
        ;
      unsigned best_score = scrabble_score(best);
      if (score > best_score) {
        cpystr(words_of_length.words[j], best);
      }
    }
  }
}
```

7.1.2 Results

We had 8 students in total participating, split into two groups. From these 7 responded to our survey.

The main improvements done by students were:

1. Recalculate the best_score only when the best word changes

2. Pass the word length as a parameter to scrabble_score to make it vectorizable

3. Use the built in memcopy or strcopy instead of cpystr

4. Pass the word length as a parameter to cpystr to make it vectorizable

Doing the first two and either the third or forth generally yielded a solution 3.6 times faster than the unoptimized version. Solutions failing to add either of the first two yielded a speedup of roughly 1.9. The students were split in half between about 3.6 times and 1.9 times the speed of the reference solution.

Because of the low number of participants and variations in how comfortable the students were programming in C we can't draw any conclusions from the raw speed ups between the control group and the one who used our tool. We don't track when and if students without the extension read optimization remarks directly from the compiler which makes the groups harder to compare as well.

Instead we walk you through two anecdotes from the student sessions coupled with their survey responses and attempt to understand how and if our tool helped them.

Azalea

Azalea is the identifier given to the student. This student was picked because they rated their comfort with C code the highest.

Log time lime:

Minute 0-5: Viewed the instructions and all optimization remarks for the functions in the given code

Minute 5-12: Edited the program to use the build in memcpy function instead of cpystr

Minute 12-13: Opened and read current optimization remarks

Minute 13-17: Played around with giving scrabble score a length parameter

Minute 17-19: Avoided recalculating the best score in every loop iteration

Minute 19-20: Added the length parameter to make scrabble_score vectorizable

Minute 20-25: Looked through the remaining optimization remarks but made no further changes

Survey response: Azalea rated themselves as very confident with the C language and was currently taking the algorithm implementation course. They found the remarks somewhat confusing and not very actionable, but they still felt they contributed to a high degree to their performance speed up. They give the "missed: regalloc" remark as an example of a confusing and inactionable remark.

In the general feedback field they wrote: "The one remark which really helped me was one about failed loop vectorization. I made scrabble_score take a length parameter and used it in the loop, and this roughly doubled the performance."

From Azalea it seems that for a programmer comfortable with C and knowledgeable about at least some important automatic optimization techniques the messages from the compiler about failed vectorizations can be useful for speeding up our code problem. The logs shows frequent use of the optimization remarks weaved together with code edits, we assume that this means the tool was useful to know what optimizations could be done by the compiler and to check if they passed in a way that was convenient enough to do it many times during coding session. Azalea claimed that they got the idea to pass a length to scrabble_score from a remark, and it might be the case that it would have taken longer to get this idea, were the remarks less available.

Ecruteak

Ecruteak was picked because this student was picked because they rated their comfort with the C language tied lowest (3/5) and had the had only read the least demanding of the relevant courses.

Log time lime:

Minute 0-6: Viewed the instructions and looked at all optimization remarks

Minute 6-12: Avoid recalculating the best score in every loop iteration

Minute 12-34: Attempt to move copy outside of the loop

Minute 34-37: Save current best word as a local variable and then remove it

Minute 37-57: Attempt to solve the issue of the failing loop vectorization, where compiler was missing the length by calling strlen inside the method. Add the comment "// clang still can't tell how many times this will run?" and then give up.

Survey response: Ecruteak rated themselves as somewhat familiar with the C language. They found the remarks very confusing and did not know what to do about even the ones they understood. They rate the number of remarks as appropriate and found the remarks to contribute to their speed up to a low degree.

An excerpt from their general feedback: "The compiler feedback was exposed well, but w/o a good understanding of what each optimization means it's still

difficult to act on. I ended up googling to try to get a better understanding of the missed optimizations, perhaps hotlinks could be provided to docs/info about the warning somewhere? (Or perhaps they exist and I missed them)."

Ecruteak indicates that they found the extension exposed compiler feedback well but that they did not know how to act on that information. From the logs we can see they also weaved saving and checking remarks together like Azalea, but without the compiler knowledge needed to actually fix the issues. From this student it seems that the tool worked well for showing that there are compiler optimizations at all and where they passed or failed.

Summary

We present two tables containing the results of the students submission7.1 and their survey responses7.2.

Table 7.1: Data from the students submissions

Student	Speed up	Minutes spent working	Revisions	In control group
Azalea	3.65	25	27	No
Ecruteak	1.97	60	187	No
Lavender	3.6	60	133	No
Rustboro	3.5	102	339	No
Blackthorn	3.61	20	64	Yes
Cerulean	2	37	118	Yes
Petalburg	0.99	12	27	Yes
Dewford	1.77	30	74	Yes

Table 7.2: Exit Survey responses

Student	Remarks confusing?	Remarks actionable?	Remarks helpful?	Comfort with C
Azalea	3	2	To a high degree	4
Ecruteak	5	4	To a low degree	3
Lavender	4	5	To a low degree	3
Rustboro	2	4	To a low degree	4
Blackthorn	1	3	Moderately	3
Cerulean	2	2	To a high degree	3
Petalburg	-	-	-	-
Dewford	4	4	Not at all	5

7.2 Performance impact

7.2.1 Method

Gzip [7] was compiled with and without remarks, with and without our various modifications to the compiler included, to measure whether each approach had a reasonable impact on the performance of LLVM and Clang. Since the source name reconstruction sometimes involves searching through all metadata in a module, larger modules may prove troublesome. Therefore we also compiled a single file version of the gzip codebase as well as a single file version of oggenc, courtesy of Stephen McCamant [20]. Gzip consisted of a total 5738 SLOC in 26 source files, as well as an additional 714 SLOC in 13 header files. The single file version of gzip consisted of 5097 SLOC. The single file version of oggenc consisted of 48063 SLOC.

Because of the large manual work involved in inserting pragmas before every single function in a codebase, pragma activated remarks were only benchmarked on the single file version of gzip. All Clang/llvm builds in the comparison were configured with `cmake -G Ninja -DCMAKE_CXX_COMPILER='/usr/bin/Clang++-10' -DCMAKE_C_COMPILER='/usr/bin/Clang-10' -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_ASSERTIONS=On -DLLVM_ENABLE_PROJECTS='Clang' -DLLVM_USE_LINKER='lld' ../llvm/`

We also collected statistics on the number of times various operations were performed in the added code, using the `-stats` flag in LLVM [17].

7.2.2 Results

Benchmark names and their meaning:

- ref: no remarks activated, baseline Clang
- allremarks ref: all remarks activated through compiler flags, baseline Clang
- optrecord ref: all remarks emitted to optimization record file,
- gvn optrecord: all remarks emitted to optimization record file, Clang augmented with our changes to the gvn remarks
- lv optrecord: all remarks emitted to optimization record file, Clang augmented with our changes to the loop vectorizer remarks baseline Clang
- sn optrecord: all remarks emitted to optimization record file, Clang augmented with our changes to the gvn remarks
- pragma flags: all remarks activated through compiler flags, Clang augmented with remark pragma capabilities
- pragma ref: no remarks activated, Clang augmented with remark pragma capabilities
- pragma activated: all remarks activated through function scoped remark pragmas, Clang augmented with remark pragma capabilities

As can be seen in Table 7.3 the remark pragma augmented compiler carries a significant penalty on compile time performance even when not emitting any remarks (compare ref with pragma ref). This penalty is still present when emitting remarks (compare allremarks ref with pragma flags), however the performance penalty of activating remarks via pragmas over compiler flags is minor (compare pragma flags with pragma activated). This logically makes sense as most of the extra work carried out by the compiler consists of checking for metadata when deciding whether to emit remarks, which is done no matter if remarks are emitted or not.

The changes to individual remarks did not have a significant impact on compile time, as shown in Table 7.4. Reconstruction of source names did have some impact, especially for oggenc. The statistics in Table 7.5 indicate that this may be due to large struct types since the type comparison has a high recursion ratio, resulting in a very large number of function calls.

7.3 Accuracy of name reconstruction

7.3.1 Method

To measure the accuracy of the reconstructed names we parsed the optimization record after compiling single file versions of gzip and oggenc. For each reconstructed name found we took the attached debug location and extracted the corresponding line in the source code to compare with the name. The special case where the debug location was set to line 0 was ignored,

Table 7.3: Performance comparison for Clang extended with pragmas for remark output, versus baseline Clang.

	gzip normal	gzip single	oggenc single
ref	1.00	1.00	1.00
pragma ref	1.12	1.17	1.07
allremarks ref	1.39	2.05	3.96
pragma flags	1.49	2.17	4.00
pragma activated	-	2.21	-

Table 7.4: Performance comparison for Clang extended with various further analyzes for remark messages, versus baseline Clang. All remarks were activated by enabling the optimization record. Normalized to optrecord ref.

	gzip normal	gzip single	oggenc single
optrecord ref	1.00	1.00	1.00
gvn optrecord	1.00	1.00	1.01
lv optrecord	0.99	1.00	1.00
sn optrecord	1.02	1.04	1.10

Table 7.5: General statistics captured by counters added to the code for source naming.

statistic	oggenc	gzip
Constructed instances of class	53066	23988
Constructed instances w/ mutable module access	50576	22506
Top level calls to getOriginalName (Algorithm 1)	86204	40275
Nested calls to getOriginalName	363096	123642
Recursive calls per call to getOriginalName	4.2	3.1
Nested calls to getOriginalName w/ dgb info return value requested	99836	10856
Instances of missing dbg info when needed	20998	8745
Bitcasts without dbg info match	13050	0
Calls to fallback method for naming fragment w/o dbg info	19945	8745
Calls to calibrateDbgType (Algorithm 14)	112186	2441
Top level calls to compareDbgType (Algorithm 7)	360141	5872
Nested calls to compareDbgType	12513962	90375
Recursive calls per call to compareDbgType	34.8	15.4

as it is not a real line in the source code. Some operations can be expressed in multiple ways, and so a conventional string comparison may indicate a mismatch between two expressions that are in fact equivalent. This, combined with the fact that naming the operations was trivial compared to the core problem of finding the right identifiers, led us to the decision to split both expressions involved in the comparison on non-alphanumerical symbols and compare each identifier in the expression with the identifier at the corresponding index in the other expression. Only expressions with expressions of equal number of identifiers were considered for matching. The debug location column number does not mark a range with start and finish, but only a single point. Because the location of this point in relation to the rest of the expression (e.g. the beginning or somewhere in the middle) is not consistent, every substring of the line was considered as a potential match and the closest one selected.

We performed these measurements for both files with and without function inlining enabled, to see whether the algorithm finding multiple aliasing identifier names due to inlining significantly affected the accuracy. To make the numbers comparable, we filtered out inlining related remarks. These remarks just contain function names anyways, which does not need to be reconstructed from metadata, and so had a nearly 100% match rate in our measurements. The number of remarks in the output still differs somewhat between the version with and without inlining, since inlining affects which optimizations can be performed.

7.3.2 Results

As can be seen in Table 7.6 and the histograms in Figure 7.1 and Figure 7.2, inlining does not have a dramatic impact on neither the mean or distribution of reconstructed name accuracies. However, they both do show a clear (if small) trend towards improved accuracy within each expression size when inlining is disabled. Note that it is impossible for an expression to match more identifiers in the source code than the number of identifiers in the expression itself, which is why the number of matching identifiers generally grows along with the size of the expression.

Dividing the matching number of identifiers with the total number of identifiers in the expression gives us a general measurement of the accuracy for identifiers. Taking the weighted mean (using the counts in Table 7.7) gives us an average accuracy of 65% with inlining for gzip, and 72% without inlining. For oggenc the values are 61% and 66%, respectively.

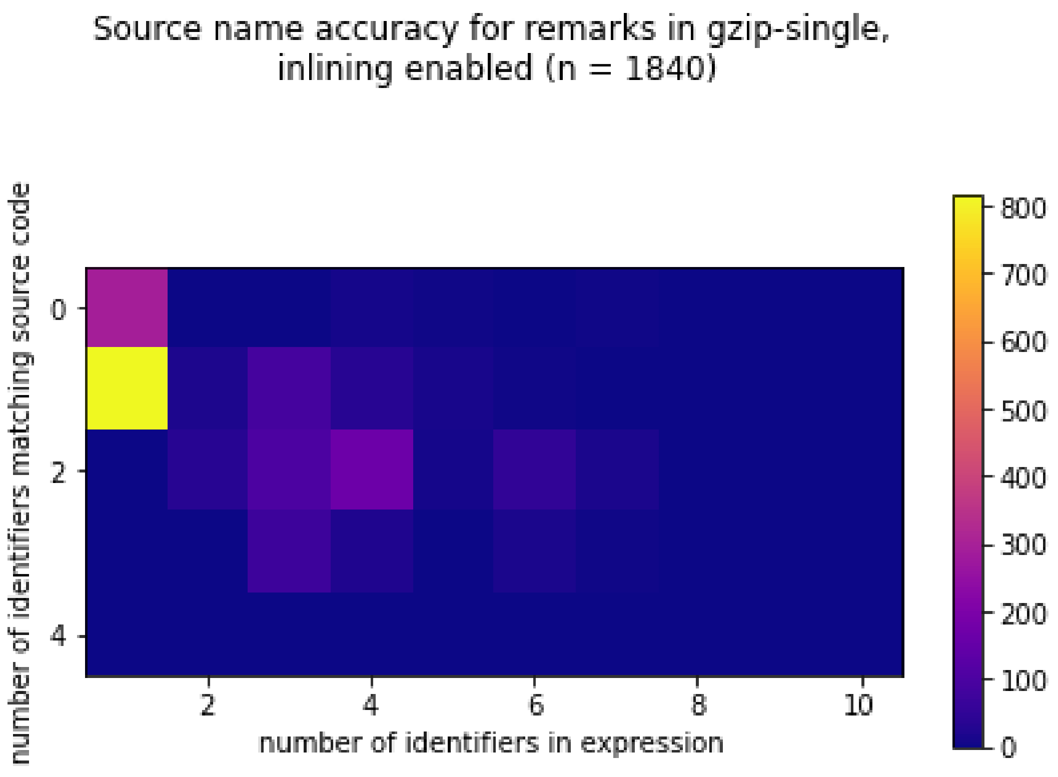
Table 7.6: Mean number of matching identifiers, grouped by number of identifiers, for reconstructed names in remarks.

number of identifiers in expression	gzip	gzip, no inline	oggenc	oggenc, no inline
1	0.735	0.832	0.613	0.663
2	1.583	1.714	1.734	1.844
3	1.923	2.130	2.067	2.206
4	1.871	1.955	2.168	2.275
5	1.394	1.667	1.774	1.903
6	2.125	2.179	1.392	1.500
7	1.667	1.846	1.162	1.258
8			1.409	1.630
9			0.571	0.818
10			0.833	-
11			0.056	0.083

Table 7.7: Remark count, grouped by number of identifiers, for reconstructed names in remarks.

number of identifiers in expression	gzip	gzip, no inline	oggenc	oggenc, no inline
1	1111	1030	1370	1107
2	60	91	1145	1054
3	272	253	1821	1571
4	241	246	1003	866
5	33	51	651	515
6	80	56	296	238
7	30	26	130	93
8			44	27
9			21	11
10			6	0
11			18	12

Figure 7.1: Histograms of name accuracy with and without inlining for gzip.



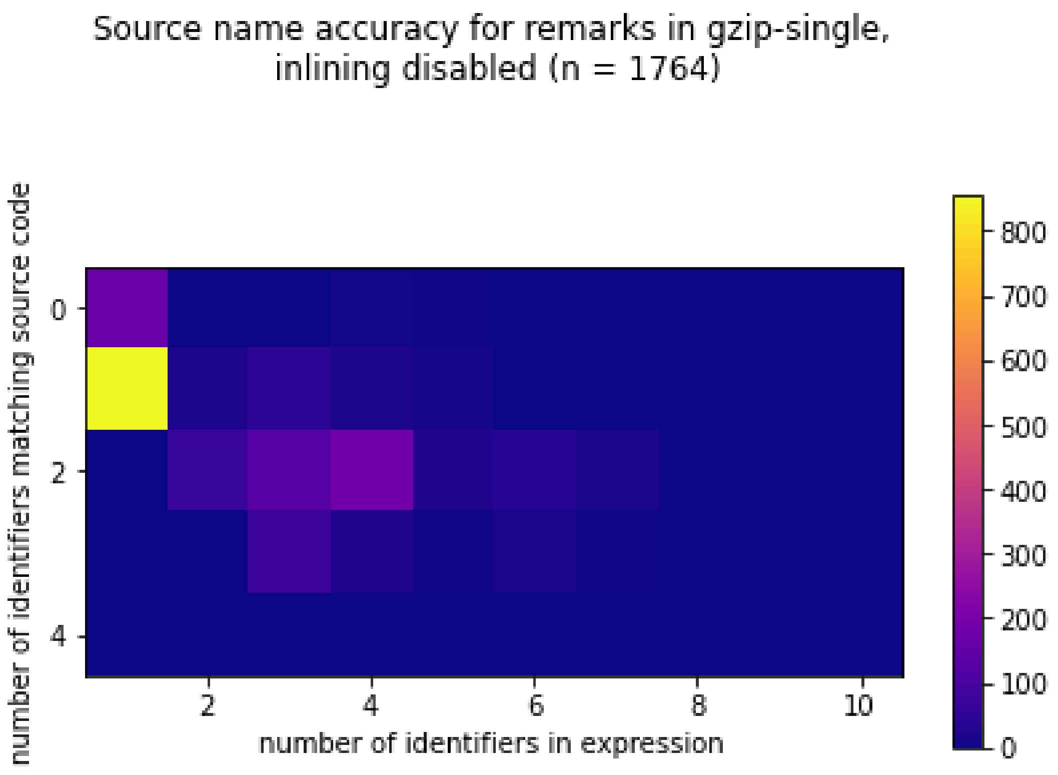
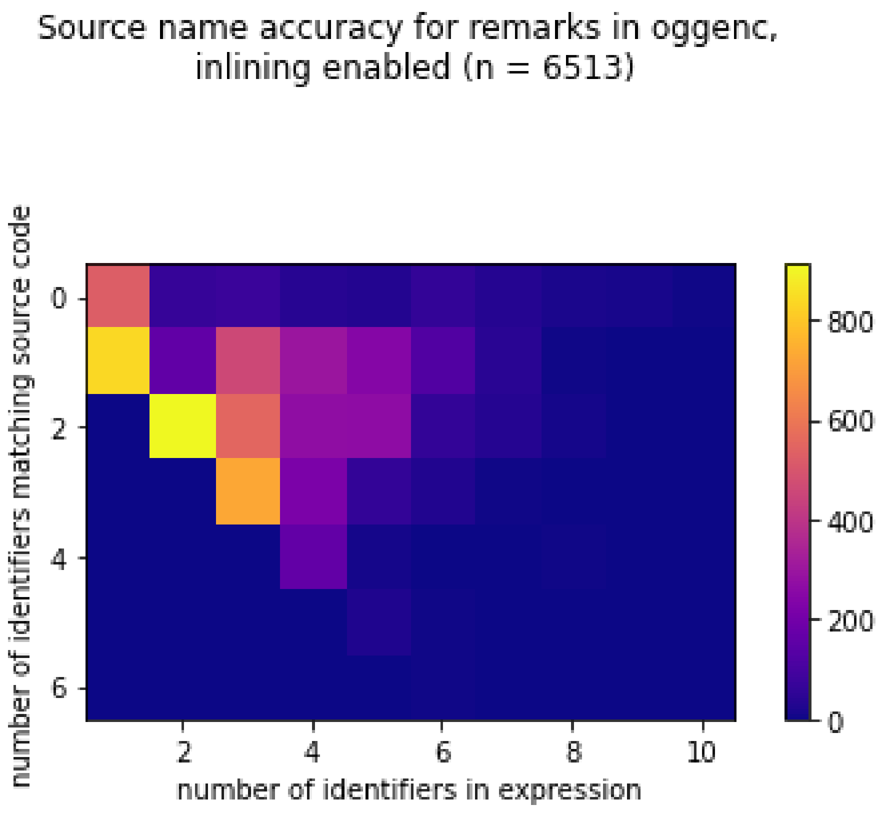
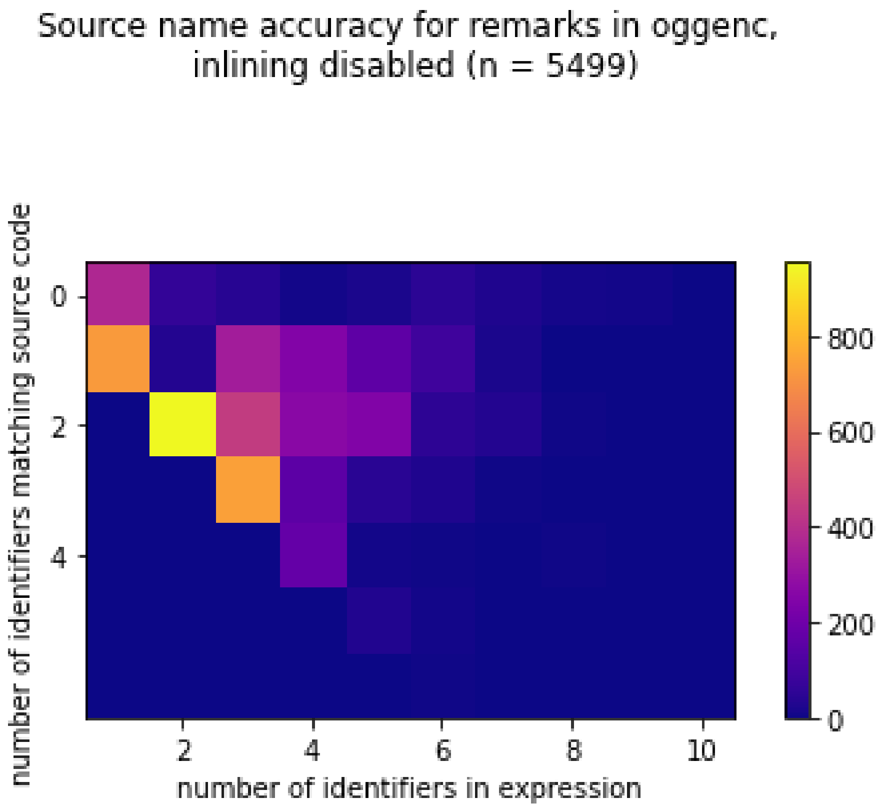


Figure 7.2: Histograms of name accuracy with and without inlining for oggenc.





Chapter 8
Related work

We introduce works related to our research questions and compare them to our work. The first paper takes a quantitative approach to see how better error messages could help programmers perform better. The second paper covers how a modified compiler can help programmers vectorize loops, guiding them through fixing the most common and fixable issues.

8.1 The usefulness of compiler error messages

To gauge the helpfulness of readable compiler error messages to students in their introductory programming course Raymond Pettit, John Homer, and Roger Gee enhanced the courses automated assessment tool [23]. They added human readable explanations and suggestions on how to fix errors on the most common errors from previous years. Several related papers used student feedback as the main indicator of usefulness, Pettit et. al wanted a quantitative approach and focused on the following measurements:

- likelihood of successive compilation errors
- occurrence of compiler errors within semesters
- student progress towards a successful completion of a programming assignment

Their results show no significant benefit to these metrics to these measurements even as anecdotal data from student surveys indicated that the tool was helpful. A qualified majority of respondents said that the messages helped them identify, fix and prevent compilation problems. Raymond Pettit et. al. speculate reasons for this contradiction. Higher achieving students are more attentive to compiler errors and are less likely to submit non-compiling code. Or newer programmers might not pay much attention to the messages from their compiler at all. They also highlight an anecdote where a student purposely submitted non-compilable code to receive better error messages after not understanding the default compiler and used the better message to get past the error. If this is a common occurrence it could also help explain the quantitative results. Since the survey-responses were small compared to the total number of participants, it's also possible that testers that used and liked the tool were disproportionately represented in the survey.

A key difference in our work is that Pettit et. al. worked with novice programmers whereas we worked with programmers with at least one year of programming experience and often more. We also have the benefit of the users actively asking for compiler messages while looking for way to improve a program, rather than them being an annoyance.

8.2 Compiler guided Optimization refactoring

Many programs contain loops that could be written in a such a way so they are automatically vectorizable but the compiler can't prove that reshaping the loops and vectorizing them won't change the behavior of the program. To allow for more loops to automatically vectorized Per Larsen et. al. [10] created an interactive compilation system that tries to hint to the programmer what changes they should consider to make the program vectorizable. This way you could use the knowledge of the programmer to make the work of the compiler easier. Their approach is to create a set of patches to the automatic parallelization subsystem of GCC [6] to emit comments back into the source code when it encounters a problem which prevents further analysis.

They then use a library they built called *libcodecomments* to make the output from GCC understandable, by generating source locations and reconstructing source level expressions and variable names from compiler created temporaries. They combine these tools with a plug-in for Eclipse C Development Tools [8] code editor to display the messages in an intuitive way directly in the source code.

Larsen et. al. use their program in two case studies of programs with a set of loops that should be vectorizable that GCC couldn't vectorize. They use their program to work around issues of aliasing, unknown loop iteration counts, complicated induction variables and data dependency issues. The now auto-vectorizable programs are benchmarked against manually parallelized versions of the same program resulting in the automatic version being better in 12 out of 23 cases.

They conclude that there are many opportunities of auto-parallelization and that for additional performance it should be combined with platform-specific tuning. They note the need for prioritization of compiler remarks since a handful of missed opportunities could result in hundreds of individual comments. In our work we allow the programmer to pick which loop or function scope and what optimizations they are interested in as a way to prevent the comments from becoming overwhelming. We did not have to create patches to LLVM to get records of loop issues since that functionality was already present.

Chapter 9
Conclusions and further work

To keep down the complexity of the code, we only construct names for derived values in C syntax. That is, the LLVM instructions are converted to the equivalent operation in C. This is still an improvement over no high-level language names, although potentially confusing to the programmer if the source code is in a language with significantly different syntax than C. Luckily, most use of LLVM is with C and other languages with C inspired syntax. In fact, it may be more precise than a language with more abstraction, where pointers cannot be accessed directly, while still keeping it more readable than LLVM IR. However, we submit that the largest benefit is of the variable names themselves and not the syntax of the derived values. As such, using a language with widespread recognition and use in the LLVM community like C should give a lot of value.

So far we have tested our modifications on C code and LLVM IR only. A quick test with C++ indicates a need to demangle function names and handle a wider range of `DIType` subtypes for use with classes. Rewriting the source naming to not construct the string directly, but instead first constructing a language agnostic AST, could make it feasible to build multiple language specific passes for constructing the string, allowing for output in the syntax of the source language instead of C syntax only.

The reconstruction of source names shows good accuracy for expressions in the range of 1-3 identifiers, which is where the majority of collected expressions fall. Our evaluation indicates that adding logic for handling aliasing identifier names in a more consistent manner may improve this accuracy somewhat. Currently aliasing identifier names are handled by simply picking one of them, but this may result in confusing remarks if names are picked from the wrong context, or - in the case of derived names - from different contexts. In the previously mentioned AST approach multiple name alternatives and their scopes could be stored, and when constructing the string effort care could be taken to be as consistent as possible in the context of the names. Alternatively, the `getOriginalName` method (shown in Algorithm 1) could have an optional `DIType` parameter for the scope of the value, and only use names available in this scope, if given. Another potential source of inaccurate names is missing debug metadata. This can be the result of either not compiling with debug symbols, or prior transformation passes being unable to preserve the metadata when transforming the IR. As an example, we found that already vectorized IR often lacked debug metadata, even when the IR in question also contained an unvectorized copy of the same code with debug metadata attached. In either way, if we do not encounter the metadata needed to name the value, we fall back to the original naming method. As such, the remarks have not lost any value, but can provide clearer references when compiled with debug information. Investigating why the accuracy for longer expressions starts to dwindle after 3 identifiers remains to be done.

To include a link with each remark to a longer explanation of the remark message has been discussed, and was suggested by a user in trials as well. This would require a large effort to document the analysis done by each pass, and to maintain this correspondence when the source code changes, but has potential to further lower the threshold for understanding among users less experienced with compiler construction and micro-optimizations. It would also improve the project's lack of documentation overall, however the general lacking in documentation may indicate that contributors are unlikely to prioritize this effort.

We also conclude that our editor integration anecdotally is a helpful tool for iterating on program performance, given that the programmer is versed in what optimizations exist and is comfortable writing C code. Measuring to see if there is any efficiency effect when using the extension and improving the extension to be distributable is left as potential future work. Our limited sample size and lack of quantitative data prevents us from answering RQ3 conclusively, from our survey responses at least the students who responded didn't perceive Clangs feedback to be effective for enabling optimizations. The same is true to for RQ2, although our results are inconclusive they seem to point towards that an increased availability of optimization remarks improve programmers' ability to speed up program performance.

Relating to RQ4 we conclude that with our current implementation, pragmas are not a good alternative to compilation flags. Pragmas for remark output negatively affect compile times significantly even when not in use. As such, the feature in its current form is not suitable for a production compiler, since a vast majority of compilations will in fact not use the feature. Potential future work may be optimizing this feature or rearchitecting it in a way that the performance impact is minimized. We also propose a more usability focused evaluation of the usefulness of pragmas compared to compile flags, as we only looked into the performance aspect. Source name reconstruction also carries a somewhat noticeable performance hit, especially for really large modules, however this is tolerable since this feature does not run any extra code unless remarks are actually emitted. As for RQ1 we conclude that the current implementation is performant enough to be usable, and the name accuracy is high enough to provide value in the added context for remarks. However there is room for improvement, for example by handling aliasing variables or mangled names. Our measurements indicate that the type comparison, and by extension the type calibration, may be a hot path when reconstructing source names. Improvements to this algorithm may reduce the performance overhead significantly. This may be even more important in the future, as LLVM is planning to transition to opaque (untyped) pointers in the IR. This could also make it worth looking into rearchitecting the debug type handling entirely, potentially avoiding the issue of type comparisons.

Improvements to individual remarks similarly only affect performance when actually used, and furthermore only affect a small subset of remarks, so the performance impact is small.

References

[1] Adam Nemet. Compiler Assisted Performance Analysis.
Presentation: `https://www.youtube.com/watch?v=qq0q1hfzidg`
Slides: `http://llvm.org/devmtg/2016-11/Slides/Nemet-Compiler-assistedPerformanceAnalysis.pdf`.

[2] Bob Haarman. Rewriting opt-viewer in C++.
`http://lists.llvm.org/pipermail/llvm-dev/2016-November/107039.html`.

[3] DWARF Standards Committee. The DWARF Debugging Standard.
`http://www.dwarfstd.org/`.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451490, October 1991.

[5] Jack W. Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable optimizing compiler. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, page 59–73, Berlin, Heidelberg, 1996. Springer-Verlag.

[6] Free Software Foundation. GCC.
`http://gnu.gcc.org`.

[7] Free Software Foundation. gzip.
`https://www.gnu.org/software/gzip/`.

[8] The Eclipse Foundation. Eclipse C Development Tools.
`http://eclipse.org/cdt/`.

[9] Coder Technologies Inc. code-server GitHub repository.
`https://github.com/cdr/code-server`.

[10] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. *Automatic Loop Parallelization via Compiler Guided Refactoring*. Number 12 in IMM-Technical Report-2011. Technical University of Denmark, 2011.

[11] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. `http://llvm.cs.uiuc.edu`.

[12] LLVM. Clang Webpage.
`https://clang.llvm.org/`.

[13] LLVM. Clangd Webpage.
`https://clangd.llvm.org/`.

[14] LLVM. LibTooling Webpage.
`https://clang.llvm.org/docs/LibTooling.html`.

[15] LLVM. LLVM Language Reference Manual.
`https://llvm.org/docs/LangRef.html`.

[16] LLVM. LLVM Loop Terminology (and Canonical Forms).
`https://llvm.org/docs/LoopTerminology.html`.

[17] LLVM. Opt Command Line Reference.
`https://llvm.org/docs/CommandGuide/opt.html`.

[18] LLVM. Remarks - LLVM 12 documentation.
`https://llvm.org/docs/Remarks.html`.

[19] LLVM. Source Level Debugging with LLVM.
`https://llvm.org/docs/SourceLevelDebugging.html`.

[20] Stephen McCamant. Single file programs.
`https://people.csail.mit.edu/smcc/projects/single-file-programs/`.

[21] Microsoft. Language Server Protocol website.
`https://microsoft.github.io/language-server-protocol/`.

[22] Stack Overflow. Stack Overflow Developer Study 2019.
`https://insights.stackoverflow.com/survey/2019#development-environments-and-tools`.

[23] Raymond S. Pettit, John Homer, and Roger Gee. Do Enhanced Compiler Error Messages Help Students? Results Inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 465–470, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

Appendices

Appendix A
Exit survey questions

I found the meaning of the optimization remarks confusing. Scale (1-5) "Strongly disagree" - "Strongly agree".

Even after understanding a remark, I did not know what to do about it. Scale (1-5) "Strongly disagree" - "Strongly agree".

I found the number of remarks. Insufficient / Less than expected / Appropriate / More than expected / Overwhelming

Insights gained from optimization remarks contributed to my speedup. Not at all / To a low degree / Moderately / To a high degree / Significantly

I understood the assignment. Scale (1-5) "Strongly disagree" - "Strongly agree".

How confident are you reading and writing C/C++ code? Scale (1-5) "Not at all" - "Very confident"

Which of the following courses have you taken part in?

- EDAF05 Algorithms, data structures and complexity
- EDAF15 Algorithm implementation
- EDAA25 C programming
- EDAN75 Optimizing compilers
- None of the above

Any other feedback you would like to share, not covered by the previous questions?

Appendix B
User evaluation task

Listing B.1: Contents of README.md

```
# Scrabble word picker
As part of a Scrabble bot you have previously written code compiling a list of ↩
    playable words for the current board.
You have now implemented a function that determines which of these words would ↩
    net you the most points.
However, while benchmarking your bot you find that finding the best word is ↩
    severely bottlenecking your overall performance.
Your task is to make the scrabble_max function finish (with a correct result!) as↩
     quickly as possible.
You may only make changes to scrabble.c.
Running `make` will compile, test, and benchmark your solution.
If you need to temporarily add flags to Clang you can do so by editing the CFLAGS↩
     variable in the makefile, however these will not be used for the final ↩
    benchmark determining the winner.
Note that the test data used does not consist of actual words.
Hint: you can view the 10 test inputs used in the benchmark in the eval/ ↩
    directory. Your solution must however also pass all the examples given in the ↩
    input/ directory.

# Submitting a solution
You have 60 minutes starting from when you first connected to VS Code.
A solution will be submitted every time you run make.
Solutions submitted after the time has expired will not be considered.
You may submit as many solutions as you want, and we will count your best ↩
    submission.
After completing the assignment please fill out this survey about your experience↩
     solving this assignment: https://forms.gle/6kKYjAMgnT9mReCe8
If you wish to participate in the competition you need to fill out your email in ↩
    the form.

# Optimization remarks
LLVM, and by extension Clang, can emit remarks to indicate whether a certain ↩
    optimization was applied or not.
The remarks come in 3 flavors:
 - pass: A remark emitted to signal that an optimization transformation was ↩
     successfully performed
 - missed: A remark emitted to signal that an optimization transformation
   could not be performed, or was not deemed profitable
 - analysis: A remark containing analysis results that can bring
   more information to the user regarding the generated code
```

```
Not all passes in LLVM emit optimization remarks, but the major ones do.
Remarks can be filtered by which pass they originated from.
Some passes that emit remarks are:
 - loop-vectorize
 - slp-vectorize
 - gvn (Global Value Numbering)
 - licm (Loop Invariant Code Motion)
 - inline

# Your tool
To help you find missed optimizations in LLVM you have a VS Code extension ↩
    installed that visualizes optimization remarks.
Whenever you save the source code file, the extension will generate remarks for ↩
    you.
You can then choose whether you want to show remarks for the whole file, a ↩
    specific function or even a specific loop, by clicking the gray text above it.
It will then let you choose which optimization pass you want remarks from, or all↩
    .

# Terminal access
You can open the terminal in VS Code by pressing ctrl+shift+c
```

Listing B.2: Contents of scrabble.h

```c
#ifndef SCRABBLE_H
#define SCRABBLE_H
struct wordlist {
  size_t length;
  char **words; // length elements
};
void scrabble_max(struct wordlist dict[], unsigned ↩
    max_word_size, char best[max_word_size]);

#endif //SCRABBLE_H
```

Listing B.3: Contents of scrabble.c

```c
#include <stddef.h>
#include <stdio.h>
#include "scrabble.h"

#define FIRST_CHAR 'A'
#define LAST_CHAR 'Z'
#define CHAR_INDEX(c) c - FIRST_CHAR
#define NBR_OF_CHARS 1 + CHAR_INDEX(LAST_CHAR)

char char_points[NBR_OF_CHARS] = {
    /*A: */ 1,
    /*B: */ 3,
    /*C: */ 3,
    /*D: */ 2,
    /*E: */ 1,
    /*F: */ 4,
    /*G: */ 2,
    /*H: */ 4,
    /*I: */ 1,
    /*J: */ 8,
    /*K: */ 5,
    /*L: */ 1,
    /*M: */ 3,
    /*N: */ 1,
    /*O: */ 1,
    /*P: */ 3,
    /*Q: */ 10,
    /*R: */ 1,
    /*S: */ 1,
    /*T: */ 1,
    /*U: */ 1,
    /*V: */ 4,
    /*W: */ 4,
    /*X: */ 8,
    /*Y: */ 4,
    /*Z: */ 10
};

unsigned scrabble_score(char *word) {
    unsigned sum = 0;
    while(*word) {
    sum += char_points[CHAR_INDEX(*word++)];
  }
    return sum;
}

void cpystr(char *origin, char *dest) {
  while ((*dest++ = *origin++))
    ;
}

/*
 * The index of 'dict' gives the list of playable words with ↩
    word size equal to that index. The 'best' buffer should ↩
    contain a copy of the optimal word when finished. In case ↩
    of a tie, the shorter word wins.
 */
void scrabble_max(struct wordlist dict[], unsigned ↩
   max_word_size,
                  char best[max_word_size]) {
  for (unsigned i = 0; i <= max_word_size; i++) {
    struct wordlist words_of_length = dict[i];
    for (size_t j = 0; j < words_of_length.length; j++) {
      unsigned score = scrabble_score(words_of_length.words[j])↩
         ;
      unsigned best_score = scrabble_score(best);
      if (score > best_score) {
        cpystr(words_of_length.words[j], best);
      }
    }
  }
}
```

Appendix C
Source name algorithms

C.1 Naming IR values

Algorithm 1 High level algorithm for reconstructing the source name of value V. A best effort is made for BasicBlocks, since these have no semantic equivalent in C source code. GetOriginalInstructionName delegates further to different subprocedures based on the specific instruction type, some of which will be described in detail. GetOriginalConstantName will not be described in detail, as it simply extracts the represented constant value.

```
procedure getOriginalName($V$)
    if hasDbgValueIntrinsic($V$) then
        $(name, dbgType) \leftarrow getNameFromDbgValueIntrinsic(V)$
    else
        switch $class(V)$ do
            case $Instruction$
                $(name, dbgType) \leftarrow getOriginalInstructionName(V)$
            case $Constant$
                $(name, dbgType) \leftarrow getOriginalConstantName(V)$
            case $BasicBlock$
                $name \leftarrow$ "$BB${"
                for Instruction $I \in V$ do
                    $(iName, \_) \leftarrow getOriginalInstructionName(I)$
                    $name \leftarrow concat(name, iName)$
                end for
                return $(concat(name,$ "}"$), null)$
            case $Argument$
                return $(argName(V), null)$
    end if
    return $(name, calibrateDbgType(dbgType, type(V)))$
end procedure
```

Algorithm 2 Algorithm for reconstructing a variable name from a debug variable intrinsic. Offsets are given in number of bits.

```
procedure getNameFromDbgValueIntrinsic($V$)
    $dbgVar \leftarrow dbgVar(V)$
    $name \leftarrow name(dbgVar)$
    $dbgType \leftarrow type(dbgVar)$
    if $isFragment(expr(dbgVar))$ then
        $nextOffset \leftarrow offset(expr(dbgVar))$
        $(fragName, dbgType) \leftarrow getVariableFragmentName(dbgType, nextOffset)$
        $name \leftarrow concat(name, fragName)$
    end if
    return $(name, dbgType)$
end procedure
procedure getVariableFragmentName($dbgType, relativeOffset$)
    switch $class(dbgType)$ do
        case $Struct$
            for $field \in dbgType$ do
                $f \leftarrow field$
                if $offset(field) > relativeOffset$ then
                    break
                end if
            end for
            $nextOffset \leftarrow relativeOffset - offset(f)$
            $(subFragName, subFragDbgType) \leftarrow getVariableFragmentName(f, nextOffset)$
            return $(concat(".", name(f), subFragName), subFragDbgType)$
        case $Pointer$
            return $(concat("[", str(relativeOffset), "]"), baseType(dbgType))$
        case $Array$
            $elemSize \leftarrow size(elem(dbgType))$
            $index \leftarrow floor(relativeOffset/elemSize)$
            $nextOffset \leftarrow relativeOffset - index * elemSize$
            $(subFragName, subFragDbgType) \leftarrow getVariableFragmentName(elem(dbgType), nex$
            $fragName \leftarrow concat("[", str(index), "]", subFragName)$
            return $(fragName, subFragDbgType)$
        case default
            return $("", dbgType)$
end procedure
```

Algorithm 3 Algorithm for reconstructing a variable name from a GEP instruction. Non-constant offsets are array/pointer offsets and named as such, while constant offsets are named based on the debug type of the value they are offset from. Offsets are actual field/element indices, not bit sizes.

```
**procedure** GETNAMEFROMGEP($V$)
    $(baseName, dbgType) \leftarrow getOriginalName(ptrOp(V))$
    $name \leftarrow baseName$
    $dbgType \leftarrow baseType(dbgType)$
    $arrayOffsetIndex \leftarrow 0$
    **while** $arrayOffsetIndex < numOffsetOps(V)$ **do**
        $arrayOffset \leftarrow offsetOps(V)[arrayOffsetIndex]$
        $name \leftarrow concat(name, "[", str(arrayOffset), "]")$
        $(constOffsets, nextNonConst) \leftarrow extractConstOffsets(offsetOps(V), arrayOffset+ 1, numOffsetOps(V))$
        $(fragName, fragDbgType) \leftarrow getValueFragmentName(dbgType, constOffsets)$
        $name \leftarrow concat(name, fragName)$
        $dbgType \leftarrow fragDbgType$
        $arrayOffsetIndex \leftarrow nextNonConst$
    **end while**
    **return** $(name, calibrateDbgType(type(V), dbgType))$
**end procedure**
**procedure** EXTRACTCONSTOFFSETS($offsets, start, end$)
    $index \leftarrow start$
    $constOffsets \leftarrow []$
    **while** $index < end \wedge isConstant(offsets[index])$ **do**
        $constOffsets \leftarrow concat(constOffsets, offsets[index])$
        $index \leftarrow index + 1$
    **end while**
    **return** $(constOffsets, index)$
**end procedure**
```

Algorithm 4 Reconstructs the name of a fragment of a composite debug type. Offsets are actual field/element indices, not bit sizes.

```
procedure getValueFragmentName($dbgType, offsets$)
    if $isEmpty(offsets)$ then
        return ("", $dbgType$)
    end if
    $offset \leftarrow head(offsets)$
    switch $class(dbgType)$ do
        case $Struct$
            $f \leftarrow fields(dbgType)[offset]$
            $(subFragName, subFragDbgType) \leftarrow getValueFragmentName(baseType(f), tail(offs$
            return $(concat(".", fieldName(f), subFragName), subFragDbgType)$
        case $Pointer$
            return $(concat("[", str(offsets), "]"), baseType(dbgType))$
        case $Array$
            $(subFragName, subFragDbgType) \leftarrow getValueFragmentName(elemType(dbgType), t$
            $fragName \leftarrow concat("[", str(offset), "]", subFragName)$
            return $(fragName, subFragDbgType)$
        case default
            return ("", $dbgType$)
end procedure
```

Algorithm 5 Algorithm for reconstructing a value name from a phi node. Naively assumes that V is an induction variable (a variable iterated through in a loop) if the debug type found matches the value type of V. Otherwise picks a predecessor known to terminate in some value with a debug variable intrinsic and uses that to name the phi. CurrentPhis is a global set used to avoid ending up in an infinite recursion: if V is in the set, the current procedure call is a direct result of an earlier call to getNameFromPhi with the same parameter.

```
$CurrentPhis \leftarrow \{\}$
**procedure** getNameFromPhi($V$)
    **if** $V \in CurrentPhis$ **then**
        **return** $(null, null)$
    **end if**
    $CurrentPhis \leftarrow CurrentPhis \cup V$
    $Visited \leftarrow \{\}$
    $Queue \leftarrow []$
    $name \leftarrow \texttt{null}$
    $dbgType \leftarrow \texttt{null}$
    **for** $Pred \in V$ **do**
        $enqueue(Queue, (Pred, Pred))$
    **end for**
    **while** $\neg isEmpty(Queue) \land name \neq \texttt{null}$ **do**
        $(n, originalPred) \leftarrow dequeue(Queue)$
        **if** $hasDbgValueIntrinsic(V)$ **then**
            $(name, dbgType) \leftarrow getNameFromDbgValueIntrinsic(V)$
        **else**
            $Visited \leftarrow Visited \cup n$
            **for** $Op \in n$ **do**
                **if** $Op \notin Visited$ **then**
                    $enqueue(Queue, (Op, originalPred))$
                **end if**
            **end for**
        **end if**
    **end while**
    **if** $name \neq \texttt{null} \land compareDbgType(type(V), dbgType) = NoMatch$ **then**
        $(name, dbgType) \leftarrow getOriginalName(originalPred)$
    **end if**
    $CurrentPhis \leftarrow CurrentPhis \setminus V$
    **return** $(name, dbgType)$
**end procedure**
```

Algorithm 6 Handling of bitcasts. Does not alter the name of the operand, but makes attempts to find a matching debug type to return.

```
procedure getNameFromBitCast($V$)
    $(name, dbgType) \leftarrow getOriginalName(op(V))$
    if $isFirstFieldNested(type(op(v)), type(V))$ then
        $(\_, dbgType) \leftarrow calibrateDbgType(type(V), dbgType)$
    else if $isFirstFieldNested(type(V), type(op(v)))$ then
        if $class(dbgType) = Pointer$ then
            $dbgType \leftarrow baseType(dbgType)$ ▷ Want to find the subtype relationship of the base types.
        end if
        $Users \leftarrow users(dbgType)$
        $Visited \leftarrow \{\}$
        $partial \leftarrow$ null
        while $\neg isEmpty(Users)$ do
            for $user \in Users$ do
                $(matchResult, calibratedType) \leftarrow calibrateDbgType(type(V), user)$
                if $matchResult = Match$ then
                    return $(name, calibratedType)$
                end if
                if $matchResult = IncompleteMatch$ then
                    $partial \leftarrow calibratedType$
                end if
                $Visited \leftarrow Visited \cup user$
            end for
            $NextUsers \leftarrow \{\}$
            for $user \in Users$ do
                for $nextUser \in users(user)$ do
                    if $fields(nextUser)[0] = user \land nextUser \notin Visited$ then ▷ Only first field uses are relevant for subtype in the cases considered
                        $NextUsers \leftarrow NextUsers \cup nextUser$
                    end if
                end for
            end for
            $Users \leftarrow NextUsers$
        end while
        $dbgType \leftarrow partial$
    else
        $dbgType \leftarrow$ null
    end if
    return $(name, dbgType)$
end procedure
```

C.2 Comparing value types and debug types

Algorithm 7 Compare value type with debug type for exact or approximate equivalence. Approximate equivalence results in the return value *IncompleteMatch*, and can represent cases such as subtype relationships, or one side having a void pointer where the other has a concrete type. Equivalent and dissimilar types have the return values *Match* and *NoMatch*, respectively.

```
procedure COMPAREDBGTYPE($valueType, dbgType$)
    switch $class(valueType)$ do
        case $StructType$
            $matchResult \leftarrow compareStructType(valueType, dbgType)$
        case $PointerType$
            $matchResult \leftarrow comparePointerType(valueType, dbgType)$
        case $ArrayType$
            $matchResult \leftarrow compareArrayType(valueType, dbgType)$
        case $IntegerType$
            $matchResult \leftarrow compareIntegerType(valueType, dbgType)$
        case $FloatType$
            $matchResult \leftarrow compareFloatType(valueType, dbgType)$
        case $FunctionType$
            $matchResult \leftarrow compareFuncType(valueType, dbgType)$
        return $matchResult$
end procedure
```

Algorithm 8 Compare struct value type with debug type. If the $\boldsymbol{debugType}$ has previously been compared with any value type without finding any inconsistencies, that value type (stored in $\boldsymbol{EquivalentStructTypes}$) must also be equivalent to $\boldsymbol{valueType}$. To handle recursive type structures, the value type must be added to $\boldsymbol{EquivalentStructTypes}$ *before* recursing on field members.

```
$EquivalentStructTypes \leftarrow ()$                    ▷ Needed for recursive types
procedure compareStructType($valueType, dbgType$)
    if $numFields(valueType) = 0$ then                   ▷ Void type, fuzzy matches anything
        return $IncompleteMatch$
    end if
    if $dbgType \in EquivalentStructTypes$ then
        $prevMatch \leftarrow EquivalentStructTypes(dbgType)$
        if $prevMatch = valueType$ then
            return $Match$
        else
            return $NoMatch$
        end if
    end if
    if $class(dbgType) \neq StructType \lor numFields(dbgType) \neq numFields(valueType)$ then
        return $NoMatch$
    end if
    $EquivalentStructTypes(dbgType) \leftarrow valueType$
    $matchResult \leftarrow Match$
    for $i \in [0..numFields(dbgType)]$ do
        $fieldMatch \leftarrow compareDbgType(field(valueType, i), field(dbgType, i))$
        if $fieldMatch = NoMatch$ then
            return $NoMatch$
        end if
        if $fieldMatch = IncompleteMatch$ then
            $matchResult \leftarrow IncompleteMatch$
        end if
    end for
    return $matchResult$
end procedure
```

Algorithm 9 Pointers can be either a pointer to a single object, or represent an array of objects, pointing to the first one. Depending on the context, a pointer representing an array can either just be of pointer type with the element type as its base type, or it can be a pointer to an array type.

```
procedure comparePointerType($valueType, dbgType$)
    if $class(dbgType) = PointerType$ then
        if $baseType(dbgType) =$ null then                                    ▷ Void pointer
            return $IncompleteMatch$
        end if
        $matchResult \leftarrow compareDbgType(baseType(valueType), baseType(dbgType))$
        if $matchResult = NoMatch \wedge baseType(valueType) = IntegerType$ then
            $matchResult \leftarrow IncompleteMatch$              ▷ Integer pointers can sometimes
represent void pointers
        end if
    else if $class(dbgType) = ArrayType$ then
        if $class(baseType(valueType)) = ArrayType$ then
            $matchResult \leftarrow compareDbgType(baseType(valueType), dbgType)$
        else
            $matchResult \leftarrow compareDbgType(baseType(valueType), elementType(dbgType))$
        end if
    else
        $matchResult \leftarrow NoMatch$
    end if
    return $matchResult$
end procedure
```

Algorithm 10 An array of longer length is a subtype of a shorter array with the same element type, so we allow them to fuzzy match.

```
procedure compareArrayType($valueType, dbgType$)
    if $class(dbgType) \neq ArrayType$ then
        return $NoMatch$
    end if
    $matchResult \leftarrow compareDbgType(elemType(valueType), elemType(dbgType))$
    if $matchResult = Match \wedge numElements(valueType) \neq numElements(dbgType)$
then
        $matchResult \leftarrow IncompleteMatch$
    end if
    return $matchResult$
end procedure
```

Algorithm 11 Enum types do not exist in the LLVM value type system, so they are represented by normal integer values.

```
procedure compareIntegerType($valueType, dbgType$)
    if $class(dbgType) = BasicType$ then
        if $size(dbgType) = size(valueType)$ then
            $matchResult \leftarrow Match$
        else
            $matchResult \leftarrow NoMatch$
        end if
    else if $class(dbgType) = EnumType$ then
        if $size(dbgType) = size(valueType)$ then
            $matchResult \leftarrow Match$
        else
            $matchResult \leftarrow IncompleteMatch$    ▷ Enum value sizes can be inconsistent
        end if
    else
        $matchResult \leftarrow NoMatch$
    end if
    return $matchResult$
end procedure
```

Algorithm 12 The basic type has a tag offering type information of finer granularity, indicating e.g. whether the value is an address, a signed integer or a floating point value. We do not currently check this tag.

```
procedure compareFloatType($valueType, dbgType$)
    if $class(dbgType) = BasicType \wedge size(dbgType) = size(valueType)$ then
        $matchResult \leftarrow Match$
    else
        $matchResult \leftarrow NoMatch$
    end if
    return $matchResult$
end procedure
```

Algorithm 13 Since IncompleteMatch does not offer information as to which type is supertype and which is subtype in cases where applicable, some functions which could be decidedly type mismatched will still end up fuzzy matched, e.g. the case where both the return type and parameter type(s) of either function type have a (strict) subtype relationship with the other function type's equivalent. This is not considered an important enough case to be in scope of this thesis, and so we are satisfied with this approximation for now.

```
procedure COMPAREFUNCTYPE($valueType, dbgType$)
    if $class(dbgType) \neq SubroutineType \vee numArgs(dbgType) \neq numArgs(valueType)$ then
        return $NoMatch$
    end if
    $matchResult \leftarrow compareDbgType(returnType(valueType), returnType(dbgType))$
    for $i \in [0..numArgs(dbgType)]$ do
        $argMatch \leftarrow compareDbgType(arg(valueType, i), arg(dbgType, i))$
        if $argMatch = NoMatch$ then
            return $NoMatch$
        end if
        if $argMatch = IncompleteMatch$ then
            $matchResult \leftarrow IncompleteMatch$
        end if
    end for
    return $matchResult$
end procedure
```

C.3 Calibrating debug type to match value type

Algorithm 14 Attempt to reach correct debug type for given value type if the debug type is slightly mismatched.

```
procedure CALIBRATEDBGTYPE($valueType, dbgType$)
    $Visited \leftarrow \{\}$                                   ▷ Avoid endless loop in recursive types
    $(result, dbgType) \leftarrow compareDbgType(valueType, dbgType)$
    while $dbgType \neq$ null $\land\ result = NoMatch$ do
        if $dbgType \in Visited$ then
            return ($NoMatch$, null)
        end if
        if $valueTypeTransitivelyPointsToDbgType(valueType, dbgType)$ then
            while $valueTypeTransitivelyPointsToDbgType(valueType, dbgType)$ do
                $dbgType \leftarrow makePointerTypeFromBaseType(dbgType)$
            end while
            return $(compareDbgType(valueType, dbgType), dbgType)$
        end if
        $Visited \leftarrow Visited \cup dbgType$
        switch $class(dbgType)$ do
            case $StructType$
                $dbgType \leftarrow fields(dbgType)[0]$
            case $DerivedType$                              ▷ Includes pointer type
                $dbgType \leftarrow baseType(dbgType)$
            case $ArrayType$
                $dbgType \leftarrow elementType(dbgType)$
            case default
                return ($NoMatch$, null)
        $(result, dbgType) \leftarrow compareDbgType(valueType, dbgType)$
    end while
    return $(result, dbgType)$
end procedure
```

Appendix D
Performance benchmark data

Table D.1: All performance benchmarks normalized by ref.

	gzip normal	gzip single	oggenc single
ref	1.00	1.00	1.00
allremarks ref	1.39	2.05	3.96
pragma ref	1.12	1.17	1.07
pragma flags	1.49	2.17	4.00
pragma activated	-	2.21	-
optrecord ref	1.03	1.07	1.05
gvn optrecord	1.03	1.07	1.05
lv optrecord	1.02	1.07	1.05
sn optrecord	1.05	1.11	1.15

Table D.2: Mean time in seconds for all performance benchmarks.

	gzip normal	gzip single	oggenc single
ref	2.55	2.3272	9.0855
allremarks ref	3.5371	4.7670	35.9443
pragma ref	2.8607	2.7144	9.748
pragma flags	3.8033	5.0501	36.3466
pragma activated	-	5.1421	-
optrecord ref	2.6196	2.4941	9.5044
gvn optrecord	2.6277	2.4900	9.5588
lv optrecord	2.6053	2.4864	9.503
sn optrecord	2.5921	2.5915	10.4378

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-08-27

EXAMENSARBETE Improving Precision and Usefulness of Clang Optimization Remarks
STUDENTER Henrik Olsson, Oskar Damkjaer
HANDLEDARE Christoph Reichenbach (LTH)
EXAMINATOR Jonas Skeppstedt(LTH)

Precise Clang Optimization Remarks Right In Your Editor

POPULÄRVETENSKAPLIG SAMMANFATTNING **Henrik Olsson, Oskar Damkjaer**

With some success we recover information lost in the Clang compiler's optimization passes to improve messages. We experiment with ways of interacting with the compiler by adding support for inline commands in the source code and by allowing interaction with the compilers optimization messages into the VSCode editor.

To improve program performance compilers automatically optimize the code they compile, meaning that for some programs they are able to keep the exact same behavior but make the program run faster. Clang is a compiler for the C-family of languages. A programmer who wants to know what optimizations were and were not applied to a program can enable so called optimization remarks using feature flags when invoking Clang. However these messages can often be abstract and overwhelming in numbers.

In our thesis we attempt to improve the experience of working with Clang's optimization remarks. The issue of the messages being abstract is partly due to the complexity involved but also because of the architecture of Clang where the names of functions and variables the programmer used are no longer present at the time of optimization. When debug data is present in the optimization pass we use this information to attempt to recreate the original names and include them in remarks where relevant. With this compiler improvement programmers would hopefully better understand what went wrong when the compiler fails to optimize a given program as the compiler is more precise.

We've tried to address the issue of optimization remarks being overwhelming in numbers and make them easier to discover in two ways. The first being adding support for writing pragmas in the source code to specify what function or loop is of interest for optimization remarks. Our experiments estimate that compilation is slowed down by 7-17% even when not emitting any remarks, purely from looking for pragmas in the metadata. Although functional, these performance issues render the feature unfeasible.

We also built a plugin to the Visual Studio Code editor. This tool annotates functions and loops and allows the programmer to pick an optimization pass right in their editor. The code is then annotated with relevant remarks. In our case study where students were tasked with optimizing a small C program, our tool seemed to speed up and improve the process of working with optimization remarks, given that the student was already comfortable with the C language. The VSCode extension could be released to the public and then easily be installed for developers who want to learn more on how Clang optimizations work.