Efficient Techniques for Points-To Analysis of C via LLVM

Mads Overgård Henningsen, 201906515
Lasse Overgaard Møldrup, 201908053

Project Report (10 ECTS) in Computer Science

Advisor: Anders Møller

Department of Computer Science, Aarhus University

June 15, 2023

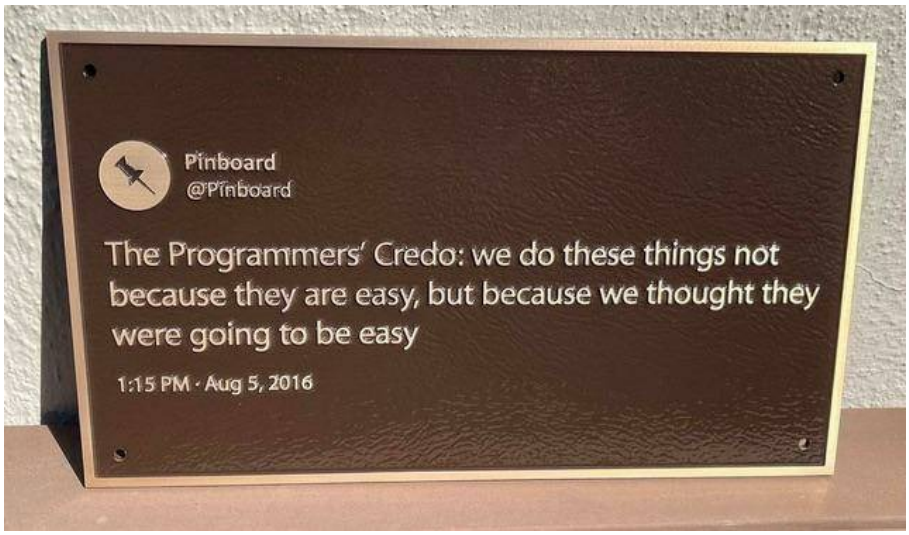
AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This report explores how to implement and optimize a points-to analysis of C. Accompanying the report is a concrete implementation of such an analysis that works by analyzing the intermediary language LLVM. We explain some of the major issues that come with using LLVM as a target for analysis and show how to handle them in practice.

The report also focuses on different techniques for optimizing the analysis. In particular, we implement the *wave propagation* algorithm by Pereira and Berlin and compare it with a basic worklist algorithm. Orthogonally, we implement and compare three term set representations: hash sets, bit vectors, and so-called Roaring bitmaps. Here, we find that the combination of Wave Propagation and Roaring term sets performs the best, in terms of both running time and peak heap memory usage. With the results, we also try to ascertain to what degree real-world points-to sets are "compact" in terms of their bit vector representation. To this question, our results are inconclusive.

Mads Overgård Henningsen and Lasse Overgaard Møldrup,
Aarhus, June 15, 2023.



Contents

1 Introduction

Points-to analysis is an important intermediary step for many other analyses, such as escape analysis. Static analyses such as points-to analysis are specifically even more important in languages that do not have a lot of static guarantees from the type system. Due to this, and its general popularity, the C language has often been studied in the literature as a target for points-to analysis [PKH07, PB09, HL07].

This report describes how one may go about implementing such an analysis for C. Specifically, we implement an Andersen-style points-to analysis tool for LLVM, the intermediary representation used by the Clang C compiler. We also discuss the pros and cons of analysing LLVM instead of C directly by describing the specific advantages and challenges that arise when C code is compiled to LLVM.

The other part of the report concerns itself with the core algorithm that solves the constraints generated in a points-to analysis. This algorithm, sometimes called the *cubic algorithm* for its running time, has also been studied extensively, and there are many optimizations in the literature that improves on the basic algorithm. We explore techniques to improve accuracy and performance of the algorithm, specifically *field-sensitivity* [PKH07], *wave propagation* [PB09], and the choice of term set representation. An experimental evaluation is made to determine the impact of these. Additionally, an experimental bitmap-based term set representation designed by us is implemented and tested. It builds on the assumption that points-to sets are "compact" in the sense that terms pointed to by the same cell are close together. We try to evaluate whether or not this assumption holds in practice.

In Section 2 we give a brief overview of LLVM as well as some notes on our notation and terminology. In Section 3, we describe the literature that we base our extensions, optimizations and choice of term set on. In Section 4, the constraint generation and solver implementations–including the implemented optimizations–are described. In Section 5, we benchmark the optimizations to evaluate their relative effectiveness. In Section 6, we give ideas for future work and touch upon some of the results and difficulties during the project.

Source code for the tool is available at https://github.com/MOH13/pointer-analysis.

2 Preliminaries

2.1 LLVM

The ultimate goal is to analyze C programs, but instead of analyzing the programs directly, we elected to analyze the intermediary language LLVM instead. LLVM is the intermediary language used by Clang when compiling C code, and it was created specifically for C and C++ with the goal of being simple to compile to assembly and to analyze. The simplicity was the main reason we chose to work with it over C. In this section, we describe the basics of how LLVM works, but skip over many details. The curious reader can find all the details of the language in [Pro23].

The type system of LLVM is similar to C with types like `i32` and `i8*` representing 32-bit integers and pointers to 8-bit values, respectively. In fact, LLVM is strongly typed, which is one of the things that sets it apart from e.g. x86 assembly. LLVM has many integer types and even represents booleans with an `i1` type. They all have the form `i`x, where x is a number, so there is no differentiation between signed and unsigned integers. Pointer types are valid when used in combination with most other types, and they generally work like in C. There are also types that are typically only found in high-level programming languages, such as the `void` type and function types. C aggregate types have corresponding types in the form of fixed-size arrays and structs that works much the same as in C. In addition, there are `vector` types in LLVM that are

semantically similar to fixed-size arrays, but are made to be used with the vector instructions that exists on all modern CPU architectures. However, C union types have no direct analogue in LLVM, but are instead handled with a combination of the other types. There are many other types in LLVM, but these are the most relevant to a pointer analysis.

The basic structure of an LLVM program is also very similar to a C program: In the global scope there are type definitions, global variable definitions, and functions. These all serve more or less the same purpose as in C. Functions consist of so-called *blocks* that in turn consist of a name (similar to a label in C), a list of *instructions*, and a *terminator*.

LLVM is an SSA (Single Static Assignment) language, which means that there is only ever one assignment to a given variable. The instructions are basic statements that work on constants and *registers*, which one may think of as an immutable local variable in another language. As an example, consider the following instruction:

```
%a = alloca i32
```

This is an `alloca` instruction that allocates space for an `i32` on the stack, and puts a pointer to that stack allocation in the register `%a`. Thus, `%a` is a register of type $i32*$. This is, in fact, the code that is generated (without optimizations) from a C-program with the statement

```
int a;
```

Note that this means that all local C variables create pointers in the corresponding LLVM code. We will discuss the implications of this in Section 6.1. Since LLVM is an SSA language, it is now no longer possible to change the value of `%a`. Now, supposed we want to use the variable `a` in the following C code.

```
a = 5;
int b = a;
```

This is converted to a `store` and a `load` instruction as follows.

```
store i32 5, i32* %a
%b = alloca i32
%0 = load i32, i32* %a
store i32 %0, i32* %b
```

Notice that the statement `int b = a;` becomes three LLVM instructions that each corresponds to one solver constraint. In general, large C expressions are broken up into many simple instructions, which is convenient for program analysis.

An important instruction for pointer analysis is the `getelementptr` instruction. Consider the following C program.

```
typedef struct {
    int f;
    int g;
} st;

int main() {
    st *st_list = malloc(sizeof(st) * 4);
    int *p = &st_list[3].g;
}
```

This program has a struct `st` with two fields `f` and `g`. It allocates a list `st_list` of 4 `st`s and then takes the address of the `g` field of the `st` with index 3. Notice that, in C, a heap-allocated list is actually a pointer to the first element of the list. Therefore, the address that we want to compute is simply an offset on this pointer. In a nutshell, such address computations is what `getelementptr` does. The above address computation could be represented with the following LLVM code.

```
%p = getelementpointer %struct.st, %struct.st* %st_list, i32 3, i32 1
```

The result of this would be that `%p` is of type `i32*`. Note that in reality `st_list` and `p` would be stack allocated, and the actual computation would therefore happen on fresh registers that are loaded from/stored to the stack. The first part `getelementpointer %struct.st, %struct.st* %st_list` signifies that we are doing a `getelementpointer` on a `%struct.st*` (the type equivalent to `st*` in C) called `%st_list`. The next argument `i32 3` means that we are offsetting the pointer by 3, i.e. it corresponds to `st_list[3]` in the C code. The final argument `i32 1` refers to the second field of `st`, because the fields are zero-indexed. In general, with nested structs, fixed-sized arrays, and vectors `getelementpointer` can take an arbitrarily long number of indices.

Terminators are special instructions that terminate blocks, as the name suggests, i.e. they are the last instruction in their block. An example is `ret` that returns from a function. Another example is `br` that branches to a different block based on a boolean value.

A global variable in LLVM is on the surface very similar to a register, only they are written as `@x` instead of `%x`. One may for instance define `@a = global i32 5`, and now `@a` can be used as an operand. The semantics of this is that `@a`, when used as an operand, is actually of type `i32*` unlike in C, which makes sense since it is modelling a mutable memory location.

2.2 Terminology and Notation

Now, we describe the notation and terminology that we use when discussing Andersen style pointer analysis. We use the words "term" and "cell" interchangeably when referring to variables, heap cells, etc. Sometimes we will write "term" to emphasize that we are referring to the contents of a points-to set. Likewise, in the context of a constraint graph, we might use the word "node" to refer to the same thing.

If t is a term, we write $[\![t]\!]$ to refer to the points-to set of t. We can then write all of the basic Andersen constraints as follows:

$$\begin{aligned} \textbf{Inclusion}: &\quad t \in [\![x]\!] \\ \textbf{Subset}: &\quad [\![x]\!] \subseteq [\![y]\!] \\ \textbf{Universally quantified}: &\quad \forall t \in [\![x]\!] : [\![t]\!] \subseteq [\![y]\!] \\ &\quad \forall t \in [\![x]\!] : [\![y]\!] \subseteq [\![t]\!] \end{aligned}$$

Note that the universally quantified constraints correspond to load and store, and they are sometimes in the literature written as $*x \subseteq y$ and $x \subseteq *y$. We will mostly refer to these constraints as load and store constraints.

3 Literature

3.1 Field-sensitivity and function pointer analysis

The straight-forward approach of having one cell per relevant variable or AST node loses a lot of precision in the cases were the node is a composite value such as a structure in LLVM or an

object in Java. In a field-insensitive analysis, the fields would effectively be merged together. There are ways to regain the lost precision. One way is to create multiple cells per variable such that there is a cell per field of the variable. Pierce et al. describe a modification to the subset-based constraints and the underlying solver to accommodate the notion of multiple cells per variable from the source code in [PKH07].

The article introduces a notion of an ordering of the cells. Cells from fields of the same variable of a composite type are next to each other sequentially in the ordering. In the case of a nested composite structure such as fields of struct type, the hierarchy is flattened. A pointer to a struct variable is represented as pointing to the first field of the variable's cells. This is similar to how pointers to a struct and its first field has the same address on many architectures. The article then introduces generalizations of the subset, load, and store constraints with the notion of an offset. To offset a term is to take its index in the ordering, add some non-negative value to it, and take the term at the new index. The notation $[\![p]\!]+i$ for a term set $[\![p]\!]$ and non-negative integer i is the set containing every term from $[\![p]\!]$ offset by i. Notice that $[\![p]\!]+0=[\![p]\!]$. We can then introduce the more general constraints $[\![p]\!]+i\subseteq[\![q]\!]$, $\forall t\in[\![p]\!]+i:[\![t]\!]\subseteq[\![q]\!]$, and $\forall t\in[\![p]\!]+i:[\![q]\!]\subseteq[\![t]\!]$. Keep in mind that for the load and store constraints it is the solution set of p that is offset and not t, i.e., it should not be confused with $\forall t\in[\![p]\!]:[\![t]\!]+i\subseteq[\![q]\!]$. In the solver, the constraint $[\![p]\!]+i\subseteq[\![q]\!]$ is represented as a new kind of weighted edge with weight i. When terms flow via an i-weighted edge, they are first offset by i.

In pseudocode, when we have some $*x=y$ where y is a composite type variable with n subfields and x is a pointer to the same type, we generate constraints $\forall t\in[\![x]\!]+i:[\![y_i]\!]\subseteq[\![t]\!]$ for all $i\in[0,n)$, where y_i is the cell for the i'th subfield of y. Intuitively, we apply store to each field separately. Likewise, a load will generate the same constraints, but with y_i and t switched.

This is sufficient for precision in most cases, but there are scenarios where precision degrades using this approach. If a positive-weight cycle, i.e. a cycle consisting of at least one positive-weight edge, is created in the graph, it will create a feedback loop that will cause the points-to sets of nodes in that cycle to explode. To mitigate this, we introduce the notion of a max offset for each term. Every term t gets some integer value $end(t)$ that is the max index it can be offset to. If offset beyond, the resulting term is ignored. Alternatively, every term can get some non-negative integer value $max(t)$ that specifies the max offset allowed on that term. The latter approach is the one used in this project. For a given term t, $max(t)$ is set to be the offset to the end of whatever composite structure the term is a part of, such that it is not possible for that term to get offset into some other unrelated cells.

The article also covers how this extension to the constraints can be used to perform a control flow analysis that takes function pointers into account. Functions can be represented as a sequence of cells, just like structs. In the beginning of the sequence, we place n return cells for an n-field return type. If the return type is not a struct, the function gets a single return cell, even if the return type is `void`. There is always only one set of return cells per function. When there are multiple return sites, all return sites flow into the function's return cells. Next, there is a cell per parameter of the function. The identifier for the function is initialized to point to the first cell of the previously mentioned sequence of cells, that is, the identifier represents a function pointer to the function. This function pointer can flow around the program like any other pointer. A function call invokes the function pointer stored in one cell with arguments stored in some list of other cells. In pseudocode:

```
d = x(a0, a1, a2)
```

where `d` is the destination variable for the return value of the function pointed to by `x` when invoked with arguments `a0`, `a1`, `a2`.

Assume that the type of `d` (and the return type of `x`) has n fields such that cells $\mathtt{d}_0, \ldots, \mathtt{d}_{n-1}$ exist, and there are exactly n return cells in the front of the sequence of the function cells. We then model flow of the return value from the called function as n constraints $\forall t \in [\![\mathtt{x}]\!] + i : [\![t]\!] \subseteq [\![\mathtt{r}_i]\!]$ for $i \in [0, n)$. Similarly, if the function is given m arguments $\mathtt{a}_0, \ldots, \mathtt{a}_{m-1}$, we model value flow into the parameters of the called function as m constraints $\forall t \in [\![\mathtt{x}]\!] + n + i : [\![\mathtt{a}_i]\!] \subseteq [\![t]\!]$.

3.2 Term set representation

There are various ways of representing the term sets of constraint variables. The obvious choice for simplicity's sake is hash sets. Implementations using hash sets are straight-forward in the sense that they exactly model mathematical sets, which is reflected in the names of types and methods. Insertion and, less importantly, deletion is near-constant time. The downside of hash sets is their relatively slow operation speed on integers when compared to specialized data structures. This is a consequence of the construction of hash sets in that they need to hash keys for insertion and deletion. Iterating hash sets is also potentially slow in the sense that the running time is bounded not by the size of the set, but by the capacity of the underlying vector.

The bitmap or bitset is a popular data structure for representing term sets. While a hash set needs more than 4 bytes to store a single `u32` term, a bitmap can use as little as 1 bit. Lookups are very fast since they are simply indexing into a byte array. Insertion and deletion are fast for the same reason. Bitmaps have the added benefit of allowing batch operations like taking the difference between two sets faster than checking single bits sequentially. This is done by performing bit-operations on the word-level. It is inefficient to represent bitmaps as a dense array of words large enough to hold each term, since if the terms are at all spread out, there will be lots of all-zero words. Various variants of compressed bitmaps exist to combat this. These are generally run-encoded [Ant95, LKA10], which results in the loss of efficient random access. The Roaring Bitmap [CLKG14] is primarily designed for usage on big data sets and uses an alternate approach of combining sorted arrays and uncompressed bitmaps in segments. These bitmaps retain efficient random access. A Roaring Bitmap is split into sequential segments of 2^{16} bits. Each segment can be represented by either (1) a sorted array of elements, (2) a dense 2^{16} bit bitmap, or (3) a run-encoding. By default, the Roaring bitmap only uses the first two representations. The sorted array representation is used when a segment contains 4096 or less elements. Otherwise, the bitmap representation is used. Roaring Bitmaps generally perform the best when their batch operations are utilized. For example, insertion of a single element is $O(n)$ in a sorted array of n elements while union of two such arrays is also only $O(n)$. Thus, it is much more beneficial to union all elements at the same time instead of inserting each individually.

In the literature, we primarily see the use of bitmaps [HT01, HL07, Nas12]. An article [Pea04] compares the usage of bitmaps ('bit-vectors' in the text), sorted arrays and balanced binary trees and finds that a hybrid of sorted arrays and bitmaps performs the best. This combination can be seen as a kind of compressed bitmap. Another article [HL07] compares the performance of compressed bitmaps and Binary Decision Diagrams and finds that BDDs are much slower than compressed bitmaps but also have a much smaller memory footprint. In C-based analyses, we generally find that GCC's compressed bitmap is used [HL07, PB09].

3.3 Cycle elimination

Edges are added to the constraint graph, when load and store constraints are fulfilled. These new edges can create zero-weight cycles, i.e. cycles consisting only of zero-weight edges, in the constraint graph. All nodes in such a zero-weight cycle will always share the same points-to set. Thus, the nodes can be collapsed to a single node by choosing any representative from the

nodes. Thus, it is possible to save a lot of redundant work propagating terms throughout the cycle.

There are various algorithms for detecting such zero-weight cycles [Pea04, HL07, PKH07, PB09]. In this report, we have implemented a variant of *wave propagation* introduced in [PB09]. The original algorithm is field-insensitive and therefore it does not consider positive-weight edges. Some small tweaks were made to maintain correctness in the presence of positive-weight edges. These will be mentioned during the overview of the algorithm following directly below.

The wave propagation algorithm consists of three phases:

1. Cycle elimination
2. Term propagation
3. Edge instantiation

These phases are repeated iteratively until a fixed-point is reached.

The cycle elimination phase is a depth-first search based SCC algorithm based on work by Nuutila and Soisalon-Soininen [NSS94]. This algorithm was slightly modified to only consider zero-weight edges in the case of field-sensitivity. This is needed to ensure that only zero-weight cycles are detected. Identified components are collapsed into a single representative. This algorithm also produces a topological order of the resulting graph when only considering zero-weight edges.

During the second phase terms are propagated throughout the nodes. All representative nodes in the graph are iterated in the (weak) topological order. For a given node, the terms added since the last iteration are propagated to its successors along all its edges. By propagating in the topological order, the amount of terms propagated are maximized. Notice that in a graph with only unweighted edges, this actually results in the maximal amount of propagated terms, leaving no terms unpropagated (before new edges are added). In the presence of positive-weight edges, some of these edges might be directed against the topological order and leave some bits unpropagated.

Lastly, if new terms have flowed into nodes that are condition nodes in universally quantified constraints, new edges are instantiated to satisfy the constraint. The original termination criteria is that there are no new edges to add in the edge instantiation phase, since the topological order results in all terms being propagated during the prior term propagation phase. This termination criteria is no longer sound, since terms can be flowed along positive-weight edges going against the topological order. As a fix, we terminate when no new terms are flowed during term propagation.

The SCC algorithm can be sped up by only performing the search on nodes that have gained new edges since the last iteration. With this approach, it is still possible to reuse the topological order from the initial run of the SCC algorithm. When a node is merged into another, it is simply removed from the order.

4 Implementation

We first describe the overall structure of the implementation. Afterwards, the details and problems we encountered are described. The source code for the analysis is available at https://github.com/MOH13/pointer-analysis.

As the programming language for the analysis we picked Rust for its speed, type system and our own familiarity. The version of LLVM that we are analyzing is version 14.0. The motivation for the architecture that we ended up with starts with how we handle LLVM programs. To parse LLVM, we use a rust crate (library) called `llvm-ir` [Dis23]. `llvm-ir` uses LLVM libraries to

parse a program given as a bitcode file (`.bc`) and then builds a data structure called a `Module` that is a full Rust representation of the program.

4.1 IR++

Even though LLVM is supposed to be a simpler representation of a program than C code, there still are a lot of complexities, as will be explained. When generating constraints, we would like to work on a simpler representation, so we introduce a new data type called `PointerInstruction`. A `PointerInstruction` abstracts away instructions that are semantically equivalent from a pointer analysis perspective into a single instruction.

The full list of `PointerInstruction`s is: `Assign`, `Alloca`, `HeapAlloc`, `Store`, `Load`, `Gep`, `Call`, `Return`, and `Fresh`. An `Assign` is a simple `x = y` assignment, but `x` and `y` can also be structs, which is encoded in the `PointerInstruction` data type. If this is the case it will then lead to multiple subset constraints being generated. The `Alloca` and `HeapAlloc` instructions represent stack allocations and heap allocations, respectively. `Store` and `Load` work in the usual way for stores and loads. A `Gep` instruction is a slight simplification of the `getelementptr` LLVM instruction where array indices are stripped away, since the analysis is array-insensitive. `Call` represents (dynamic) function calls that have the option of having a destination. The `Return` instruction signifies that a variable is being returned (and thus should be assigned to the return cell of the function). In the analysis, we emit cells for every left-hand side of a `PointerInstruction`, which intuitively should mean that every variable gets emitted. There are, however, special cases where we wish to create a cell, but not assign anything to it, e.g. when handling a `null` constant. As is the case with `Assign`, we encode the (possible) struct types of the operands in the `PointerInstruction`, whenever it is appropriate.

An example is the `bitcast` and `addrspacecast` LLVM instructions that convert from one pointer type to another and they are thus implemented as an `Assign PointerInstruction`. `PointerInstruction`s also strip away a lot of unnecessary attributes and type information as well as arguments to instructions that are irrelevant.

This conversion happens on-the-fly, i.e. a new whole program representation is not generated, but constraints are instead generated for `PointerInstruction`s immediately. Another benefit is that the conversion does not have to be one-to-one either. We can totally ignore instructions that work on e.g. integers by not emitting any `PointerInstruction`s for them. This then automatically ensures that no cells are included in the solver from these irrelevant instructions. As will be explained later, this also allows us to break down complex LLVM constructs into a series of `PointerInstruction`s.

4.2 Architecture

The architecture is illustrated in Figure 1. The basic usage is to construct a `PointsToAnalysis` and call the `run` method on a `Module` generated by `llvm-ir`. This gives a `PointsToResult` that can be used to get the points-to sets. The details of how `PointsToResult`s are structured does not matter for this explanatino. A goal of the project is to compare how different constraint solver implementations perform. Therefore we have a `Solver` trait (similar to interfaces in object-oriented languages) that represents a constraint solver, which will be explained further in Section 4.4. The `run` method is therefore generic in the `Solver` that should be used to solve the constraints. Solvers are initialized with a list of cells and allowed offsets as explained in Section 3.1. Therefore, we need to traverse the program twice, where we in the first run use a `PointsToPreAnalyzer` that collects those cells and allowed offsets and also finds how many cells should be in each heap allocation, i.e. the size of the biggest struct. The generation and solving

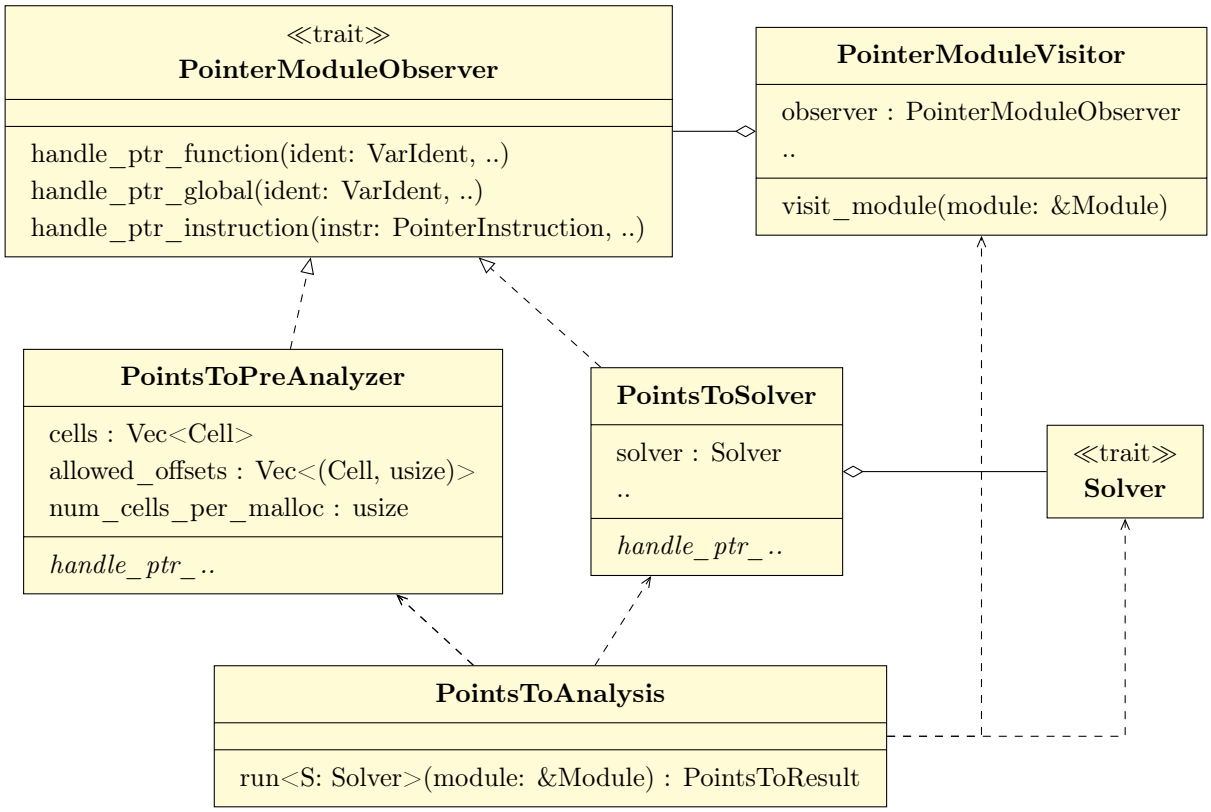


Figure 1: The architecture of the implementation in UML. Note: The diagram is presented in an object-oriented style with dynamic dispatch, but in reality generic type parameters are used.

of constraints is then done in a second traversal of the program with a `PointsToSolver`.

As mentioned, we wish to analyze our simplfied IR++ representation, but we also need to run the `PointsToPreAnalyzer` on this version of the program. This is the motivation for having the `PointerModuleVisitor` and corresponding `PointerModuleObserver` trait. A `PointerModuleVisitor` visists all functions, global variables and instructions of a LLVM `Module` and emits IR++ versions for a `PointerModuleObserver` to handle. As an example, if the `PointerModuleVisitor` sees an LLVM `load` instruction, it calls `handle_ptr_instruction` on its observer with a `Load PointerInstruction`. Seeing as both `PointsToPreAnalyzer` and `PointsToSolver` implement `PointerModuleObserver`, we can run two separate instances of the visitor, which gives the desired effect.

4.3 Constraint Generation

We now explain the details of constraint generation. There are five kinds of cells:

- Var cells: Represent registers, global variables and functions.
- Stack cells: Allocation site abstraction for the `alloca` instruction. Note: stack allocations of local variables seem unlikely to happen in loops, since in C, variable declarations can be hoisted out of loops, thus the allocation site abstraction of the stack should be very precise.
- Heap cells: Allocation site abstraction of heap allocations. These are generated for C standard library functions like `malloc` and `calloc`.
- Global cells: Represent the actual memory locations of global variables, as opposed to the variables pointing to those allocation.
- Return cells: All return values flow into these cells. Needed for function pointers to work.

Let us look at a simple C program that illustrates the different cell types.

```
int global_var = 1;

char *alloc_byte() {
    char *heap_var = malloc(sizeof(char));
    return heap_var;
}
```

A slightly simplified LLVM-version of the program looks like the following.

```
@global_var = global i32 1

define i8* @alloc_byte() {
entry:
    %heap_var = alloca i8*
    %malloc_call = call i8* @malloc(i32 1)
    store i8* %malloc_call, i8** %heap_var
    %ret_val = load i8*, i8** %heap_var
    ret i8* %ret_val
}
```

Going top down, the first line `@global_var = global i32 1` generates two cells: A Var cell representing the `i32*` variable `@global_var` that can be referenced directly in the code, and a Global cell representing the memory location pointed to by `@global_var` that can only be referenced with indirection through `@global_var`. An inclusion constraint would then be generated making the Var cell point to the Global cell. Next, the function `@alloc_byte()` that returns an `i8*` is defined. A Var cell is generated for the function pointer itself, which then points to a Return cell of the function. The function has no parameters, but these would have Var cells if there were any. The first three instructions correspond to the first line of the C function, and the last two correspond to the last line in C. The first (`alloca`) instruction creates a Var cell for `%heap_var` and a Stack cell for the stack allocation pointed to by `%heap_var`. This also means that an inclusion constraint is generated. The second (`call`) instruction creates a Var cell for `%malloc_call`, and because `malloc` is a specially handled standard library function, a Heap cell and inclusion constraint is generated. The third instruction that is a `store` does not create any new cells, but simply generates a universally quantified constraint. The fourth (`load`) instruction creates a Var cell for `%ret_val` and generates a universally quantified constraint. Finally, the `ret i8* %ret_val` instruction does not generate any cells, but creates a subset constraint going from the `%ret_val` Var cell to the Return cell of the function.

4.3.1 Aggregate Types

The implemented analysis is array insensitive, which means that LLVM arrays are represented by a single cell. Therefore, the analysis treats all indices of the array as one. This is a trade-off between precision and performance/simplicity. The hope is that the contents of an array is uniform enough that this is a reasonable abstraction. We have chosen to handle LLVM vectors the same way.

In order to work with structs in our intermediary representation we need to treat them differently from pointer values. This is because one LLVM struct value becomes multiple cells, which means that we may need to generate multiple constraints. Concretely, for all interesting values in the LLVM program we have identifiers that then become cells. The reason why we make this distinction is among other things that we can have a base identifier that represents a struct value, but does not itself generate a cell. Instead, there are so-called offset-identifiers built on top of other identifiers that represent fields. We also introduce a `StructType` which represents the type of a (nested) struct. We can then give an optional `StructType` in all relevant `PointerInstruction`s, which can then work on the appropriate offsets.

Consider the following innocent-looking C program:

```
struct_t a, b;
b = a;
```

This program is converted to something like

```
%a = alloca %struct.struct_t
%b = alloca %struct.struct_t
%0 = bitcast %struct.struct_t* %a to i8*
%1 = bitcast %struct.struct_t* %b to i8*
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %0, i8* %1, i64 ..)
```

That is, for a simple C assignment, the compiler strips away the type information by casting the struct pointers to `i8*` and then uses an LLVM intrinsic called `llvm.memcpy.p0i8.p0i8.i64` that copies bytes between the two pointers. Since our analysis is field-sensitive, we need to copy

each field one-by-one, i.e create subset constraints for each field. This is a case where we have to rediscover the C code that leads to the LLVM code, since the intrinsic doesn't carry the required type information to do this assignment properly. Our solution is to maintain a map that maps pointer registers to their original struct type. As an example, all of the register in the above example are mapped to `%struct.struct_t`, since they are either a direct allocation of a struct or is a `bitcast` of a struct pointer. When handling the call to `llvm.memcpy.p0i8.p0i8.i64` we can then look up `%0` and `%1` to find the appropriate offsets of field that should be copied (the types should match). Since we are dealing with pointers to structs instead of just structs, we create a fresh cell for each field, which we then can load into and store from.

Another issue is to handle what happens when a struct is allocated on the heap. The problem is that a call to `malloc` always returns a `void*`, which means that we do not know how many cells to allocate for it. Our solution is to pre-compute the size of the largest struct in the program. We then create enough cells to fit that largest struct, which is sound since any extra cells do not cause problems. While this is not ideal, there is no reliable general way to know what struct is being allocated as the argument to `malloc` is just a size in bytes. This is done in a "flat" manner, i.e. only the size–not the structure–of the largest struct is preserved. This is fine, since we are only accessing the fields on the heap in an indirect manner, which does not care about their structure, as we will explain shortly.

To access a field of an allocated struct in LLVM, the `getelementpointer` (`GEP`) instruction is used. When used on a pointer to a struct, `GEP` can be used to get a reference to one of the fields on the struct, as described in Section 2.1. In case of nested structs, `GEP` can even get sub-elements of a field. `GEP` can also in a completely equivalent manner index into possibly nested LLVM arrays.

The `GEP` instruction is translated into an offset that corresponds to the field that the `GEP` indexes to, following the order that the cells were generated in. A `GEP` on a term p resulting in an offset k which is assigned to term q is represented as the constraint $[\![p]\!] + k \subseteq [\![q]\!]$.

As mentioned in Section 3.1, it is possible, even with simple and realistic code, to generate a positive cycle. If there are no bounds to how much a cell can be offset, then this will degrade analysis precision. This is caused by the fact that a reference to the last cell of a struct can get offset to be a reference to something else entirely. We fix this by introducing a max offset max_i to each cell C_i. Most cells will have a max offset of 0, that is, they cannot be offset at all. Cells belonging to a struct will have a max offset such that they can only get offset to the end of that struct. The constraint $[\![p]\!] + k \subseteq [\![q]\!]$ then means that $\forall c, c \in [\![p]\!] \wedge k \leq max_p \implies (c + k) \in [\![q]\!]$.

4.3.2 Constant Expressions and Global Variables

In LLVM, many instructions can take *constant expressions* as arguments instead of simple operands or constants (e.g. `i32 1`). As an example the `add` instruction could add together the registers `%x` and `%y`, but it could also add `%x` with the constant expression `add i32 2, i32 3`, i.e. `x + (2 + 3)`. This is usually not a problem for a pointer analysis, since constants tend to be ignored in pointer analyses. That is, until you consider global variables. The trouble is that global variable references are actually constants and so must be implemented differently from registers. Furthermore, they may appear in arbitrarily complex constant expressions that can include multiple such global variable references.

Those instructions that can appear in constant expressions are those regular LLVM instructions that do not have side effects, e.g. `add` and `bitcast`. Thus, we would like to treat them in the same way as the corresponding regular LLVM instructions, by generating `PointerInstructions` for each. To achieve this, we make fresh identifier cells that hold the intermediary values of the sub-expressions, so they are of the same SSA form as the regular instructions. This also includes

struct and array initializers, which must be handled in a special way. We have seen real world examples of large global initializers of arrays of structs that need to be condensed to single structs in our analysis due to the aforementioned array-insensitivity.

4.4 Solvers

The solver architecture was designed to be modular so that it is possible to replace the solver with little configuration needed in the constraint generation and analysis phase. As a result, all solvers use the same analysis frontend, minimizing code duplication and maintenance overhead. This helped us multiple times when bugs were found in a solver. The term set representation (hash set, bitmap etc.) was abstracted as the `TermSetTrait` trait so that the solver was decoupled from the underlying term set. Thus, we can test how each term set representation perform together with each solver type. `TermSetTrait` is defined as follows:

```
pub trait TermSetTrait: Clone + Default {
    type Term;

    type Iterator<'a>: Iterator<Item = Self::Term>
    where
        Self: 'a;

    fn new() -> Self;
    fn len(&self) -> usize;
    fn contains(&self, term: Self::Term) -> bool;
    fn remove(&mut self, term: Self::Term) -> bool;
    fn insert(&mut self, term: Self::Term) -> bool;
    fn union_assign(&mut self, other: &Self);
    fn extend<T: Iterator<Item = Self::Term>>(&mut self, other: T);
    fn difference(&self, other: &Self) -> Self;
    fn iter<'a>(&'a self) -> Self::Iterator<'a>;
}
```

The function `new` creates a new empty term set. The functions `len` returns the number of term present in the set. `contains`, `remove` and `insert` are the basic single-term methods to check for inclusion, remove and add to the set. All terms present in the set can be iterated using the `iter` function. There are also functions that operate on whole term sets at a time. The `union_assign` function performs the in-place union of two term sets. The `difference` function creates a new term set with the difference of the two provided sets. Implementations of `union_assign` and `difference` can be much more efficient than simply handling terms individually, most notably for the bitmap-based representations.

Integers are the obvious choice for terms in solvers since they can then be identified by their index in the ordering required by field-sensitivity. A top-level solver called `GenericSolver` was created that operates on `Cell` terms that is used by the analysis. The solver does not perform much itself, but translates the `Cell`s to integers using a simple lookup. The actual constraint solving is then delegated to an underlying solver that operates on integer terms.

For the basic implementation, a single solver, the `WorklistSolver`, and term set implementation for hash sets was created. The `WorklistSolver` is a worklist-based solver with `u32`

terms. The `HashSet` term set is implemented using the `hashbrown` [d'A23] crate's `HashSet` and `HashMap` implementations that performs better than the default `std::collections::HashSet` and `std::collections::HashMap`.

4.5 Optimizations

A `WavePropagationSolver` was created based on the wave propagation algorithms described in [PB09]. Minor adjustments were made to the algorithm to support field-sensitivity as described in Section 3.3. Care was taken to properly utilize the batch operations supported by bitmaps. For example, differences are propagated via a union operation instead of each term being individually added to the target term set. Terms propagated via positive-weight edges are added individually instead of in batch because of implementation details. To efficiently unify nodes during the SCC phase, a set of reverse edges is maintained for each node. A reverse edge exists if and only if some normal edge of any weight exists in the opposite direction. This reverse edge map is used to efficiently identify which incoming edges need to point to the new representative of the node instead. Care must be taken to ensure that the reverse edge map is up to date with the state of actual edges to ensure ensure correctness.

Two term set implementations based on bitmaps were made. The first is a Roaring Bitmap-based term set implemented using the `roaring` [Ren22] crate. The second bitmap implementation, simply called `BitVec`, is a custom solution made to explore alternate bitmap-based term set representations. This bitmap stores the term set in an uncompressed bitmap in a word-aligned interval. That is, the size of the bitmap is determined by the distance between the lowest and highest bit present in the bitmap. The intuition was that points-to sets are not random and are generally 'clustered' with more references to variables in the same function or module than outside. In that case, these bitmaps might be reasonably effective. If this scenario does not hold, they are probably very inefficient. We will explore this in the following section.

5 Evaluation

In this section we define a set of research questions regarding the performance and efficiency of the analysis:

- **RQ1:** How does the wave propagation solver perform in comparison to the basic worklist solver?
- **RQ2:** How does the choice of term set representation affect performance and memory usage of the two solvers?
- **RQ3:** Are real-world points-to sets compact and localized and if so/not, how does it affect the performance and memory usage specifically for the `BitVec` term set representation?

A comparative benchmark of the different optimizations implemented was performed. These tests were carried out on an AMD Ryzen 7700 with 32 GB of RAM running in WSL 2 on Windows 11. Table 2 compares the running times of different solver implementations on five different programs. The solvers are either of the "Worklist" or "Wave" variety, referring to the worklist and wave propagation algorithms described in the previous section, respectively. For both of these, each of the following three term set representations are tested: "Hash", which uses hash sets, "BitVec", which uses our own bit-vector implementation, and "Roaring" which uses the roaring bitmap crate from [d'A23]. We ran each solver/program combination ten times and computed the mean running time. For added context, the problem size (i.e. the number of

	Num. terms	Inclusion constraints	Subset constraints	Univ. cond. constraints
Curl	51646	2531	9007 (1123)	5403
Make	20573	1932	10472 (1250)	9594
Htop	49930	6221	16448 (4811)	25688
Vim	356007	22170	187456 (38275)	173353
Emacs	374976	15989	195356 (30526)	141496

Table 1: Problem sizes for each program. Number in parentheses under 'Subset constraints' indicate how many of the edges were of positive weight.

	Curl	Make	Htop	Vim	Emacs
Worklist Hash	0.020 s	3.533 s	46.38 s	DNF	DNF
Worklist BitVec	0.018 s	10.33 s	116.5 s	DNF	DNF
Worklist Roaring	0.020 s	16.54 s	183.4 s	DNF	DNF
Wave Hash	0.024 s	0.352 s	2.808 s	90.20 s	OOM
Wave BitVec	0.031 s	0.375 s	4.127 s	153.5 s	616.0 s
Wave Roaring	0.031 s	0.212 s	1.712 s	58.89 s	246.5 s

Table 2: Comparison of the running times of different solver implementations for five different programs.

terms and constraints) for each program can be seen in Table 1. Some runs did not finish within two hours (DNF) and another program ran out of memory (OOM). We also measured the peak memory usage of the analysis for each combination in separate runs. Table 3 shows the results of this.

Let us look at **RQ1**. We see that memory usage and running time is better for the Worklist solver than the Wave solver for the small program Curl. However, it scales much worse than the Wave solver, to such a degree that it did not finish analyzing the two largest programs. The Wave solver might be better than the Worklist solver for two reasons: Cycle elimination and batch operations. Cycle elimination removes nodes from the problem graph and reduces both the overhead of propagating and the memory usage of points-to sets. Batch operations theoretically allow for much more efficient propagation of terms. We do note however, that the Hash term set does not support batch operations. Despite that, the Wave/Hash combination performs comparably in running time to the Wave/Roaring combination. This might indicate that cycle elimination in itself is a large contributor to the performance gains. It might be relevant to investigate if some Worklist solver with cycle elimination would perform well. We suspect that the Hash term set would perform better with such a solver than with the Wave solver.

	Curl	Make	Htop	Vim	Emacs
Worklist Hash	31.27 MiB	170.5 MiB	542.0 MiB	DNF	DNF
Worklist BitVec	33.54 MiB	152.8 MiB	521.7 MiB	DNF	DNF
Worklist Roaring	34.44 MiB	255.4 MiB	913.4 MiB	DNF	DNF
Wave Hash	36.31 MiB	120.1 MiB	409.7 MiB	15.95 GiB	OOM
Wave BitVec	45.05 MiB	93.20 MiB	364.4 MiB	18.31 GiB	21.03 GiB
Wave Roaring	39.93 MiB	69.83 MiB	179.57 MiB	4.80 GiB	14.23 GiB

Table 3: Comparison of peak heap memory usage of different solver implementations for five different programs.

Regarding **RQ2**, we see that the effect of the term set choice depends very much on the choice of the underlying solver. Roaring is the best term set both running time and memory wise for the Wave solver, while it performs much worse with the Worklist solver. The reason seems to be that the sorted array representation used by Roaring when there are less than 4096 elements in a segment is not particularly efficient when using single operations as opposed to batch operations. As a (previously mentioned) example, insertion in such an array is $O(n)$ where n is the amount of elements in the array. As of now, we do not have an explanation of the larger memory usage of the Roaring term set with the Worklist solver. In general, we see that the Hash term set performs well with both solvers running time wise. In the Wave solver, its memory usage is much higher than with the Roaring term set however.

Lastly, we will consider **RQ3**. In theory, the BitVec term set could perform better than the other term sets if the assumption of compact and localized term sets is true. In practice, we see it performs worse than both of the other term sets. We do see however, that the BitVec term set is more memory efficient than the Hash term set in a few instances. Most notably, the Wave/BitVec combination did not run out of memory on Emacs. This might indicate that the assumption held sufficiently during the analysis of that specific program. In general, the BitVec term set seems to be more unreliable in its efficiency when compared to the other term sets. This can be attributed to its reliance on the order of the terms. As a future endeavour, it could be interesting to explore other possible orderings of the terms.

An interesting observation that is not covered by the research questions is that Curl is a significantly bigger program than Make when measured by terms with a comparable number of constraints. Yet, analyzing Curl is much faster and takes much less memory. The reason for this is that the points-to sets computed for Make are much larger than Curl. In fact, Make produces many identical points-to sets of size 271. Investigating this with a visualization of the constraint graph, it turns out that Make, Htop, Vim, and Emacs all contain large cycles, with and without positive edges. As alluded to earlier, cycles cause loss of precision. The wave propagation solver is able to collapse the zero-weight cycles (cycles without positive edges), but the positive cycles are still a problem. They especially cause problems due to our heap cells of which there are a number equal to the size of the largest struct for each allocation. These cells can be offset, which means that a positive cycle will multiply them. This can be seen in Htop, where there are extremely large points-to sets that contain almost all fields of multiple heap allocations. This issue also artificially increases the density of the points-to sets, which could call into question the validity of our conclusion to **RQ3**.

6 Discussion and Future Work

6.1 Discussion of C vs LLVM

The thought was initially that it would be easier to implement an analysis of LLVM than of C. This thought was somewhat challenged by our experiences during the course of this project. While there certainly are some advantages to analyzing LLVM, the disadvantages are also numerous. We will list some of our thoughts and observations here.

We will start with the advantages. By analyzing LLVM as opposed to C, we might achieve a degree of portability, making it easier to analyze other languages that compile to LLVM. This would require some kind of frontend for each language however. We had to make workarounds to properly support certain C features like struct assignment (see Section 4.3.1). Similar and maybe more extensive workarounds would probably be required for other languages.

Another advantages is that the analysis of LLVM can utilize optimizations performed higher up in the tech stack during compilation of the source language. Any simplifications of the LLVM output could simplify the analysis and even improve the precision, for example when dead-code elimination is performed. However, optimizations might also remove variables that the end user is interested in, making it impossible to give them relevant results. Utilizing optimizations in the tech stack can also be seen as a bit of a philosophical cheat, since the performed optimizations might themselves depend on some pointer analysis.

We will now cover some disadvantages. When LLVM was chosen initially, we thought that LLVM would have a simplified structure when compared to the C source code. In particular, we had hoped that the SSA form of variables and the generally normalized form of instructions would simplify analysis. While some simplifications could be made as a result of the SSA form, the existence of constant expressions (see Section 4.3.2) greatly negates the apparent normalized instruction form. Even though constant expressions do not appear in the majority of the lines of the code, it is still vital to support this language feature. Any reasonably sized real program will compile to LLVM that depends on some particularly nested and mildly terrifying constant expressions. Ignoring these will degrade the accuracy of the analysis greatly.

An obvious disadvantage is that LLVM code is larger than code from its source language and introduces more identifiers that in turn add more cells and constraints. Even "just" a 50 percent increase in the problem size is problematic, since the theoretical running time of Andersen pointer-analysis is cubic. It might be possible to counteract this loss in efficiency by performing some simplifications of the constraint graph based on properties of LLVM. For example, it is not possible to point to an LLVM register. This could be exploited by collapsing these nodes in the graph before it is solved.

Another problem is that useful abstractions often disappear during the compilation from the source language to LLVM. This is the case with struct assignment, where we had to implement a workaround to regain the information lost during compilation. LLVM also introduces a lot of complexity and low level detail such as alignment of data and various flags that only become relevant when the LLVM is eventually compiled to machine code. This added noise led us to implement the IR++ language layer (see Section 4.1) to regain a useful level of abstraction for pointer analysis. The hope was initially that LLVM itself would serve as this layer to the source language. Since this was not the case, once can wonder if it had been more productive to implement this abstraction language directly on top of C.

Lastly, it is not always easy to associate variables or allocation sites in C with corresponding cells in the LLVM analysis. Additionally, these associations or even the general structure of the LLVM program might change as a result of seemingly small changes in the C source code.

6.2 Future work

In the constraint generation there are still a lot of C standard library functions that are not yet implemented, and LLVM features that the analysis is not sound for. An example is the `inttoptr` instruction that, as the name suggests, converts an integer to a pointer. One could track when integers are converted to pointers with the `ptrtoint` instruction and then have soundness if pointers are simply converted to integers and back. General soundness for this instruction would have to include giving up and adding all terms to some points-to sets, however.

In the same vein, handling the use of casting between pointer types along with pointer arithmetic (translated to `getelementpointer` instructions), which allows indirect access of struct fields, is another area that the analysis is not sound for.

It would also be interesting to test the analysis on other languages that compile to LLVM. We have not investigated how much work would be required to do so, but given that the analysis does have some knowledge of C features (e.g. with struct assignments), it would probably require some extra work.

Another direction would be to try to fix the blow-up of the points-to sets seen in almost all of the benchmark programs. As explained, this happens due to positive cycles in the programs. A possible remedy would be context sensitivity, which would hopefully lead to fewer new edges being added in the analysis, which would hopefully remove the positive cycles. Lei and Sui also give a new technique that merges terms that would otherwise blow up in positive cycles in [LS19]. Essentially, the technique is based on the observation that a positive cycle of weight k increases the offset of terms with multiples of k. Furthermore, this technique is compatible with both the basic constraint solver and wave propagation. It would be interesting to implement this and measure its impact, and add it to the comparison

Likewise, another recent improvement comes from [LLSH22], which introduces Pus (Partial Update Solver), a completely novel solver that sees large speedups over wave propagation. Pus sees the largest improvement on context-sensitive pointer analysis, but also improves on context-insensitive analysis. The core idea of the algorithm is to compute a much smaller *causality sub-graph* in each iteration that is sufficient to propagate the new points-to information. Pus is also compatible with the technique from [LS19], so it would be interesting to see how much these two improvements together could speed up the analysis.

7 Conclusion

The goal of this project was to implement an Andersen-style points-to analysis of C using LLVM and to test the performance of variants and optimizations of this analysis. This was done by implementing a generic constraint generator that can be combined with any solver. Additionally, each solver can be combined with any term set. A baseline worklist solver (Worklist) and a potentially more optimized wave propagation solver (Wave) was implemented. Additionally, 3 term set representations were implemented: Hash, BitVec and Roaring.

The running time and memory usage of the 6 combinations of solver and term set were tested on 5 C programs of various sizes. Some of these programs had positive cycles in their graph representation, which caused them to have large points-to sets and comparatively long running times. With the Worklist solver, the Hash term set performed the best. This could be attributed to the fact that batch operations cannot be utilized in the Worklist solver. On smaller programs, the Worklist solver performs comparably to or better than the Wave solver but it quickly fell behind on both running time and memory usage. Generally, the Wave/Roaring combination had the shortest running time and lowest memory usage. The other term sets had 2-3 times slower running time and 4-5 times higher memory usage in combination with the Roaring solver. We

conclude that this advantage is caused by performant batch operations and good compression in the Roaring bitmaps.

In conclusion, we found that the wave propagation solver using the Roaring compressed bitmaps had the best performance compared to the baseline Worklist solver. Supplements to and alternatives to the wave propagation algorithm exist in the literature and could possibly improve performance even further.

References

[Ant95] G. Antoshenkov. Byte-aligned bitmap compression. In *Proceedings DCC '95 Data Compression Conference*, pages 476–, 1995.

[CLKG14] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *CoRR*, abs/1402.6407, 2014.

[d'A23] Amanieu d'Antras. hashbrown (version 0.13.2) on crates.io, 2023. Available at https://crates.io/crates/hashbrown/0.13.2.

[Dis23] Craig Disselkoen. llvm-ir (version 0.9.0) on crates.io, 2023. Available at https://crates.io/crates/llvm-ir/0.9.0.

[HL07] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 290–299, New York, NY, USA, 2007. Association for Computing Machinery.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. PLDI '01, page 254–263, New York, NY, USA, 2001. Association for Computing Machinery.

[LKA10] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, jan 2010.

[LLSH22] Peiming Liu, Yanze Li, Bradley Swain, and Jeff Huang. Pus: A fast and highly efficient solver for inclusion-based pointer analysis. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1781–1792, 2022.

[LS19] Yuxiang Lei and Yulei Sui. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings*, page 27–47, Berlin, Heidelberg, 2019. Springer-Verlag.

[Nas12] Rupesh Nasre. Exploiting the structure of the constraint graph for efficient points-to analysis. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, page 121–132, New York, NY, USA, 2012. Association for Computing Machinery.

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[PB09] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *2009 International Symposium on Code Generation and Optimization*, pages 126–135, 2009.

[Pea04] David Pearce. Some directed graph algorithms and their application to pointer analysis. 2004.

[PKH07] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1):4–es, nov 2007.

[Pro23] LLVM Project. Llvm language reference manual, 2023. Available at https://llvm.org/docs/LangRef.html.

[Ren22] Clément Renault. roaring (version 0.10.1) on crates.io, 2022. Available at https://crates.io/crates/roaring/0.10.1.