NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Designing Decoupled Compiler Transformation APIs

Stefanos I. Baziotis

Supervisor: **Yannis Smaragdakis,** Professor

ATHENS

July 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεδιάζοντας Αποσυνδεδεμένες Διεπαφές για Μετασχηματισμούς Μεταγλωττιστών

Στέφανος Ι. Μπαζιώτης

Επιβλέπων: **Γιάννης Σμαραγδάκης**, Καθηγητής

ΑΘΗΝΑ

Ιούλιος 2021

BSc THESIS

Designing Decoupled Compiler Transformation APIs

Stefanos I. Baziotis
S.N.: 1115201600105

SUPERVISOR: **Yannis Smaragdakis,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σχεδιάζοντας Αποσυνδεδεμένες Διεπαφές για Μετασχηματισμούς Μεταγλωττιστών

Στέφανος Ι. Μπαζιώτης
Α.Μ.: 1115201600105

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Σμαραγδάκης,** Καθηγητής

ABSTRACT

The fundamental design of optimizing compilers has not changed for many decades. It is oriented around passes, each of which tries to apply a specific transformation. A celebrated benefit of this design is the separation of concerns, because each pass is concerned with a single transformation. But what is subtle is that it also hinders the separation of concerns.

In modern instances of this design, each pass has at least two responsibilities: performing a transformation and deciding whether the transformation is profitable. Inevitably, the code dealing with each of these responsibilities is tightly coupled and one can't interface with or change each piece separately.

However, each of these responsibilities is radically different and the people having the expertise to improve one do not necessarily have the knowledge to even understand the other. For instance, profitability could be picked up by a machine-learning researcher, who is interested in improving the heuristics used but is not at all interested in learning how the compiler performs the transformation.

In this thesis, we present a prototype of a transformation, implemented over the LLVM framework, used to serve a broader goal; that of separating transformations in their own independent, granular APIs. Such APIs should offer high-control and minimal cognitive load to the user, whether this is a compiler expert or not.

The transformation we chose is loop distribution because it is easily comprehensible yet potentially highly effective, while its implementation is comparable to LLVM's upstream version.

SUBJECT AREA: Compiler Optimization

KEYWORDS: compilers, transformations, API, loop distribution, LLVM

ΠΕΡΙΛΗΨΗ

Ο θεμελιώδης σχεδιασμός μεταγλωττιστών που έχουν ως κύριο στόχο βελτιστοποιούν τον κώδικα δεν έχει αλλάξει εδώ και δεκαετίες. Προσανατολίζεται γύρω από τα περάσματα (passes), κάθε ένα από τα οποία προσπαθεί να εφαρμόσει ένα συγκεκριμένο μετασχηματισμό. Ένα περίφημο όφελος αυτού του σχεδιασμού είναι ο διαχωρισμός των αρμοδιοτήτων, επειδή κάθε πέρασμα ασχολείται με έναν και μόνο μετασχηματισμό. Αλλά αυτό που ίσως είναι δύσκολο να διακρίνουμε είναι ότι επίσης υπονομεύει τον διαχωρισμό των αρμοδιοτήτων.

Στα σύγχρονα παραδείγματα αυτού του σχεδιασμού, κάθε πέρασμα έχει τουλάχιστον δύο ευθύνες: να πραγματοποιήσει έναν μετασχηματισμό και να αποφασίσει αν ένας μετασχηματισμός είναι επικερδής. Αναπόφευκτα, ο κώδικας που ασχολείται με κάθε μία από αυτές τις ευθύνες είναι στενά συνδεδεμένος και δε μπορεί κάποιος να έχει διεπαφή ή να αλλάξει κάθε κομμάτι ξεχωριστά.

Ωστόσο, αυτές οι δύο ευθύνες είναι ριζικά διαφορετικές και οι άνθρωποι που έχουν την τεχνογνωσία να βελτιώσουν τη μία δεν έχουν απαραίτητα ούτε καν τη γνώση να καταλάβουν την άλλη. Για παράδειγμα, το πρόβλημα του κέρδους ενός μετασχηματισμού μπορεί να αναληφθεί από έναν ερευνητή μηχανικής μάθησης ο οποίος ενδιαφέρεται να βελτιώσει τις ευρεστικές αλλά δεν ενδιαφέρεται καθόλου να μάθει πως ο μεταγλωττιστής φέρνει εις πέρας τους μετασχηματισμούς.

Σε αυτή την εργασία, παρουσιάζουνε ένα πρότυπο ενός μετασχηματισμού, υλοποιημένο πάνω στο σύνολο εργαλείων LLVM, το οποίο χρησιμοποιούμε για να προάγουμε έναν ευρύτερο στόχο. Αυτός ο στόχος είναι να απομονώσουμε τους μετασχηματισμούς στις δικές τους ανεξάρτητες και με μεγάλο βαθμό λεπτομέρειας διεπαφές. Τέτοιες διεπαφές θα πρέπει να προσφέρουν μεγάλο έλεγχο και ελάχιστο γνωστικό φορτίο στο χρήστη, είτε αυτός εξειδικεύεται στους μεταγλωττιστές είτε όχι.

Ο μετασχηματισμός που επιλέξαμε είναι η ανακατανομή βρόγχων (loop distribution) επειδή είναι εύκολα κατανοητός ενώ είναι δυνατόν να αποδειχθεί εξαιρετικά χρήσιμος. Η υλοποίησή του είναι συγκρίσιμη με αυτή της έκδοσης που χρησιμοποιείται αυτή τη στιγμή από τους χρήστες του LLVM.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βελτιστοποίηση Μεταγλωττιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: μεταγλωττιστές, μετασχηματισμοί, αφηρημένες διεπαφές προγραμματισμού, κατανομή βρόγχων, LLVM

ACKNOWLEDGEMENTS

I am wholeheartedly grateful to my supervisor, Dr. Yannis Smaragdakis. He was always available, armed with sharp comments and eager to help me get through it during a turbulent period. Furthermore, I would like to thank Dr. Simon Moll, my main collaborator during the time when the ideas of this thesis evolved and took shape. He was the first person to turn my attention towards their value and usefulness.

CONTENTS

LIST OF FIGURES

PREFACE

The ideas we present in this thesis were born, took shape and blossomed during my time as a compiler researcher in NEC Deutschland GmbH. There, I worked closely with Simon Moll and we needed a tool that just transforms loops and does not deal with legality or profitability (we wanted to deal with those two separately). Moreover, we wanted to be able to use that tool any time we wanted during a compilation. The standard steps when one wants to program a transformation is to just create a pass. But, transformation passes can't be called at will from inside the compiler. Because of that, they cannot just transform code. They have to deal with all prerequisites of a transformation; namely, legality and profitability. Thus, it became apparent that a pass-oriented design would not do the trick.

I was excited to attack this problem and try to build an alternative infrastructure. What followed was a deep dive into API design patterns. Fortunately, I was lucky enough to have come across of Casey Muratori, who I consider a major figure in API design (he is celebrated in Section 2.1). Furthermore, progressively I have been thinking more and more about what benefits can we gain from decoupling transformations and profitability heuristics. This exploration is still underway, but some premature thoughts and ambitious hopes are scattered throughout this text.

1. INTRODUCTION

Consider Fig. 1.1.

```
1 for (int i = 0; i < n; ++i) {
2   a[i+1] = a[i] + b[i];
3   d[i] = e[i] + f[i];
4 }
```

Figure 1.1: Loop-carried Dependence

We can see that the first iteration writes to **a[1]** and the second iteration reads from it. This implies that the second iteration depends on a result written by the first one, which enforces an order between the two. We *must* execute the first iteration *before* the second, so that the written data is available when the second iteration arrives. Similarly, the third iteration depends on the second, the fourth on the third and so on, resulting in a chain of dependences which ultimately enforces us to execute all the iterations in order.

This type of (data) dependence is called ”loop-carried” because data is carried from one iteration to another. Loop-carried dependences are unfortunate for parallelization aficionados because they prevent them from executing their iterations in parallel. Sadly, usually we cannot avoid loop-carried dependences because the serial execution they impose is ingrained in the very nature of the computation at hand. It’s like eating in a restaurant; you cannot start eating before the cook has prepared your food and this dependence is inherent in the food preparation process.

Nevertheless, the impact of such dependences can be diminished by at least not executing the rest of computations in the original order. For instance, in Fig. 1.1, if we could delete line **2**, each iteration of this loop would be independent of each other, which would let us execute them in parallel. Well, we cannot delete it but what we can do is extract it from this loop and put it on its own, like in Fig. 1.2.

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

for (int i = 0; i < n; ++i) {
  d[i] = e[i] + f[i];
}
```

Figure 1.2: Distributed Loop

Extracting parts of a loop into their own loop falls under the loop distribution transformation, which simply divides a loop in multiple ones. It comes in handy when trying to break dependences but this is not its only use case. Maybe we want to parallelize some parts of a loop, while we want to vectorize some others. Or maybe we want to unroll just a fraction of it and ideally, we would like this fraction to be in a separate loop.

Given its usefulness, it is no surprise then that loop distribution finds its place in LLVM, a popular optimizing compiler. However, its implementation is less than ideal.

Loop distribution in LLVM is implemented as a pass. Passes in LLVM, much like in any other conventional compiler, have as their purpose to do a pass over the code and apply a specific transformation. For example, one pass might be responsible for eliminating dead code, that is, code which is useless, while another pass might be responsible for replacing a call to a function with the body of the function itself.

This design has many benefits, one of them being the separation of concerns. Each pass is responsible for applying one and only one transformation, while it allows us to compose multiple of such passes to achieve a complex transformation result.

But, it is not always profitable to apply a transformation even if it is possible. For instance, it is usually not profitable to replace all function calls with their bodies, as it is usually not profitable to distribute all loops. This implies that we have to somehow take profitability into account when we apply a transformation.

Consider for example again the case of loop distribution in Fig. 1.1. Let's say that we do a pass over this code with loop distribution in mind. We have to perform two actions. The first is deciding whether distributing this loop is profitable and which instructions we should extract. This involves reasoning like "I see a loop-carried dependence, so maybe I can put that in its own loop, which will enable parallelization". In a different loop, the reasoning for splitting it might be different e.g., "these parts of the loop should be vectorized while those should parallelized so let's split them apart".

If the first action decides that we should in fact extract some instructions, a second action is triggered. This action is responsible for actually changing the code and it involves "creating a new loop, copying some instructions to the new loop, deleting these same instructions from the original loop" etc.

The point here is that the code which implements the first action and that which implements the second are *wildly* different. Not only do they serve distinct purposes, but also their implementation requires a different arsenal of knowledge.

Ideally, then, we would like these two actions to be clearly separated within a compiler. However, this is not usually the case. The parts of the code performing each of the actions are usually entangled together, in such a way that it is nearly impossible to change one without having to understand its deep connection with the other. And so, if a programmer is only interested in changing the profitability code but does not have any knowledge of the transformation code (neither does she care or should care to have), they are going to face big problems.

The main purpose of this thesis is to provide a decoupled loop distribution API, implemented over the LLVM infrastructure, whose sole purpose is to perform the loop distribution transformation. A secondary purpose is to design this API so that it gives as much control as possible to the user. Effectively, this API gives users the ability to split loops, by choosing which instructions they want to extract and having a lot of control in which steps of the transformation they want to perform and when.

More broadly, it serves as an example to promote a greater idea; that of separating transformations in their own, independent libraries. Such a design enables a novel separation of concerns. Those concerned only with profitability can program it independently and interface with the transformations when they want to, without having to understand their internals.

2. BACKGROUND

2.1 The Principles of API Design

We designed the loop distribution API based on the principles initially presented in [8]. In the original video, the presenter bases his discussion on game development problems, explanations, examples and solutions. Nonetheless, we believe that these design principles are not game-development-specific and in fact are powerful for general API design. To that end, we wish to bring them to the attention of the compiler developers and so, we will include a short description for each of them with more generic rationale and examples.

2.1.1 Granularity

An API is granular depending on how much it gives the user the ability to separate a step into smaller steps that achieve the same thing. As an example, consider Fig.2.1.

```
// Task: Allocate and zero N bytes of memory
// Using one big step: calloc()
void *m = calloc(N);

// Using two smaller steps: malloc() and memset()
void *m = malloc(N);
memset(/* base pointer */ m, /* byte to set */ 0,
       /* how many bytes */ N);
```

Figure 2.1: Allocating Zeroed Memory

The reason granularity is useful is because it gives control to the user. Maybe they want to complete some of the steps at a later point or interleave some steps in between and a granular API gives them this ability.

On the other hand, if an API does not provide sufficient big steps, then it might require a lot of boilerplate to use it when the user is only interested in the canonical cases.

This is especially true during the bootstrapping period of a project, which consists of the initial moments of a project, where the programmers usually want to test high-level ideas. This period does not require much control and so users are fine with, and actually desire, the big steps of an API.

On the contrary, when a project has progressed a lot and all the elaborate details have been discovered, the user of an API wants as much control as possible over it so that he can adapt it to his exact needs.

In short, the trade-off of granularity is *simplicity vs flexibility*.

2.1.2 Redundancy

An API exposes redundancy when the user can accomplish the same thing in different ways. The first usual case is directly related to granularity. If an API provides granularity, then it is usually redundant too because it gives the user multiple options as to where they will use small or big steps.

The other common case is when the user can pass data to the same utility in different forms. As an example, let's say that we want to provide an API which calculates the Body Mass Index (BMI) of a person. To compute this, we need the height and the weight of the person.

In our program, we might have a data type **Person** representing a person, which *among other things*, contains the height and weight.

Let's say that the BMI calculator API provides two function calls, in Fig. 2.2.

```
float calculate_BMI(Person p);
float calculate_BMI(float height, float weight);
```

Figure 2.2: Two Alternatives to Calculate the BMI

This API gives us some redundancy on how we can calculate the BMI. We can either pass a **Person** or we can pass directly the height and weight.

Redundancy is another way to provide control. The usage code of the API may have a **Person** available on some places or individual heights and weights on others. If the API provided only the first call, the latter places would have to pack the height-weight pairs in a **Person** to be able to call the calculator. If on the other hand the API provided the second call, the former places would have to *unpack* the height and weight from a **Person**.

However, we also have to consider that the more redundancy an API delivers, the less orthogonal it becomes. Orthogonality is the exact opposite of redundancy. An API is orthogonal when there is a single way to accomplish anything. Orthogonality can be desirable because it simplifies the usage of the API in multiple ways. First, the user does not need to stress over which way is the best to use the API because there is only one. Second, the user only needs to understand this one way. When an API is used across a code base, in a non-orthogonal API, the user has to be able to understand and keep in mind all the possible ways in which the API can be used.

So, the trade-off in redundancy is *orthogonality vs convenience*.

2.1.3 Retention

Retention exists when the API forces the user to announce to it data *she owns* and it keeps some form of a copy.

A classic example is **strtok()**. You give it a string and a delimiter and then, you call it continuously to give you back the extracted strings of this specific string that you passed on the start. So, **strtok** *retains* the original string. This makes the API unusable in the case where e.g., the user wants to tokenize two strings in parallel.

Another not so obvious example of retention is the **malloc() / free()** combination. When the user allocates memory with **malloc()**, he gives a size and he (hopefully) gets back a pointer. When he wants to free the memory, he calls **free()** but giving it *only* the pointer. However, most **free()** implementations can benefit from knowing the initial size when freeing the memory [2]. It should be obvious that there is no general way of saving both a memory address and the size in a single pointer (which is what is passed to **free()**), which means that the allocator has to somehow *retain* the size.

In general, retention is useful because it lets the API do a lot of things automatically. In the first example above, the user does not need to keep intermediate state that **strtok()** possibly needs to work and give it back to it in every call. In the **malloc() / free()** example, the user simply does not need to keep track of the size.

However, retention can be bad for both the user and the API implementer. On the one hand, it might enforce the user to synchronize their ”world” and course of action with that of the API (just as with the **strtok()**). On the other hand, it may make the implementation of the API difficult, as with the **malloc() / free()** interface. This interface has made the implementation of allocators notoriously hard.

In short, the trade-off of retention is *synchronization vs automation*.

2.1.4 Flow-Control

Flow-control refers to whether the API ever *calls back* the user. In the case in which this is not happening, if we imagine the call stack, then there is a point from which any functions called beyond it are only API functions, like in Fig. 2.3.

```
------------- Call Stack -------------
F
E
D
--------------- Anything below: user code.
                Anything above: API code
C
B
A
```

Figure 2.3: Call Stack

The avoidance of call-backs gives the user more control over when everything is called, and so, over the execution flow. If the user has to e.g., provide some kind of function pointer to the API, the user now loses the ability to control when this function is called.

2.1.5 Coupling

Coupling exists when some action A implies B. Coupling is sort of the bad side of everything mentioned above.

When the API is not granular enough, it means that you have one big step and you cannot break it into smaller steps. That means that if you want to do only one of the small actions contained in this big step, then it is implied that you have to do all of the rest. One prominent form of coupling is the **new** operator in C++, which couples allocation with initialization. It took a long time to C++ to provide a decoupled version of this API, using what is called *placement* ***new***.

When the API is not redundant enough, there might be utility functions you want to use but they accept only a specific type. So, calling these utility functions implies that you have to convert to this type.

And of course coupling can come in higher-level ideas. An obvious example is the transformation and cost-modeling in modern compilers. They cannot be *de*coupled and use each one separately.

Coupling is pretty much always bad and should be avoided when possible. Albeit, sometimes it is not possible due to many reasons, one possible being the nature of the problem.

2.2 LLVM

2.2.1 The Rise of LLVM

LLVM [6] started out as Low-Level Virtual Machine and the goal was to ship programs in a form of a target-independent intermediate representation (IR), called LLVM IR. Having programs in this form, one could do lifelong optimization and keep using newer and better compiler technology on this IR. Reality struck and that goal was abandoned due to hardware definitions changing over time and the inability of LLVM IR to remain hardware-agnostic. However, LLVM survived, to say the least, as a highly successful optimizing compiler back/middle-end.

A compiler can be thought as a box taking a high-level source language on one end, usually called the front-end, and produces assembly at the other, usually called the back-end. In modern compiler designs, the front-end is not just a conceptual end, but a separate software piece, performing part of the total work. The same goes for the back-end, which performs the rest of the work.

The front-end still takes a high-level language but its job ends in translating it into a more primitive (i.e., low-level) representation. This representation is usually called intermediate representation (IR) because it sits in the middle of two ends. Then, the back-end takes the baton (which is the IR in this case) and continues the work to eventually produce (ideally optimized) assembly.

One core benefit of having two ends communicating using this low-level IR, instead of a single monolithic piece, is that such an IR can express multiple high-level languages. For instance, whether we are talking about C with its relatively close-to-the-machine attitude or C++ with its crazy lambda/class/concept/template features, both can be boiled down and expressed using simple operations like loads, stores, adds and jumps. So, instead of creating a compiler for each language, we only create a piece of it, the front-end, all of which translate to the same IR. Because in this way, we only need to create *one* compiler to handle the the rest of the pipeline; one which takes IR and produces assembly.

Similarly, in the back-end, a lot of optimizations are target-independent. If a statement is useless, it is useless whether we eventually aim to produce x86 or AArch64. So, removing such statements is a target-independent optimization. In that regard, and because the IR is conceptually target-independent, we perform such optimizations in the IR, instead of programming this transformation for each different assembly.

These benefits are the main reason LLVM survived and succeeded, through the use of LLVM IR. As mentioned, LLVM IR is not exactly target-independent, but the fragments that are not constitute edge cases. And because LLVM IR is so well-designed and well-specified, it became the target of multiple front-ends. Suddenly, anybody who wanted to create language didn't need to sweat writing an optimizer in order to get decent executables. They just needed to write a front-end that translates to LLVM IR. The rest was left to (the) LLVM (back-end).

2.2.2 LLVM IR

LLVM IR [3] is for the most part well-designed, well-specified, well-understood and widely used. However, it is also huge and complex. Furthermore, the problems that this thesis tries to solve along with their solutions are not specific to LLVM or LLVM IR. Essentially, LLVM acts as a means to an end and it is not the end itself. So, for the purposes of this thesis, the reader only needs to have a rudimentary understanding of LLVM IR.

The main LLVM IR piece of reference is a module. A module contains functions, functions contain basic blocks and basic blocks contain instructions. Among those terms, a ”basic-block” is probably the most unfamiliar. It is simply an ordered list of non-branching instructions, one following the other, which eventually follows a branching (a.k.a. terminator) instruction. For our purposes, a branching instruction is going to be a branch or a return instruction. A branch is simply a jump instruction which jumps to a different part of the code and can be conditional or unconditional.

An important characteristic (or lack, thereof) of LLVM IR is that loops are not first-class objects. This means that there is no entity to represent a loop in LLVM IR. In other words, a parser and semantic analyzer which parses LLVM IR is oblivious to any loops that might exist. Loops are simply by-products of the way LLVM IR is written and the user has to discover them on their own (e.g., the user has to write code which finds whether there is a basic block which we can visit multiple times, thereby implying a loop).

There is already some infrastructure that automatically finds loops for us and saves their characteristics, structure and other information in **LoopInfo**; we will come across **LoopInfo** later. If the reader is interested in learning more about loops in LLVM, we recommend the official loop terminology of LLVM [4] (which is in fact more than terminology and almost a tutorial).

2.2.3 LLVM Middle-End and Core Infrastructure

Target-independent optimizations are not a small part of the work of an optimizing compiler. Actually, the are so important, that now compilers, and especially LLVM, are thought of having a distinct piece for target-independent optimizations, called the middle-end. A slight digression here to mention that the name middle-end does not make much sense, simply because nothing ends in it. But I guess the popularity of the terms ”front-end” and ”back-end” made the establishment irresistible.

Anyhow, the LLVM middle-end comes with a handy core infrastructure which lets the LLVM developer represent and manipulate LLVM IR using C++ code. For instance, the **llvm::Instruction** C++ class represents an LLVM IR instruction and it contains information about it like its opcode. As a convention, we use the notation **llvm::<some type>** to make clear that this is not a type of our own and instead belongs to the LLVM types of the core infrastructure.

A characteristic of the LLVM types is that they are organized around inheritance relationships. For example, **llvm::Instruction** inherits from a more generic type, **llvm::Value**, which represents any value that can appear in LLVM IR code. But also, for each specific instruction, there is a different specialized sub-class which inherits from **llvm::Instruction** and contains possibly additional fields and methods. For example, the **llvm::LoadInst** C++ class represents an LLVM IR **load** instruction.

Except **llvm::Instructions**, the only other type which is of particular interest is the **llvm::Loop**. This is just the representation of a loop discovered in LLVM IR (remember that loops in LLVM IR are not first-class objects and they have to be discovered by a separate analysis).

2.3 Dependences

Dependences that can arise in a program are not a central part of this thesis. Here, we will provide an overview of what one needs to know to understand the content we present later. For a full discussion on the topic, we refer the reader to [5].

2.3.1 Control and Data Dependences

If an instruction **A** is control-dependent on an instruction **B**, then **B** controls whether the execution will reach **A**. As an example, in Fig. 2.4, the instruction which implements the condition **if (a)** controls whether the instruction **b = 2** will be executed.

```
if (a) {
  b = 2;
}
...
```

Figure 2.4: Control Dependence

If an instruction **A** is data-dependent on an instruction **B**, then **B** affects the data that **A** operates on. For example, in Fig. 2.5, the instruction **a = 2** affects the data used in instruction **b = a + 2**.

```
a = 2;
b = a + 2;
```

Figure 2.5: Data Dependence

We should note that data dependences can be more subtle. In Fig. 2.6, we do not know whether the pointers **p** and **q** alias. If two pointers alias, it means that they may access overlapping memory. For instance, if **p** was set to point to **&a** (where **a** is a variable for the sake of this example) and **q** was set to point to the same address, then the two pointers alias.

If two pointers alias, then writing to the memory pointed by the one affects the memory pointed by the other since by definition these two memory ranges overlap. We do not know if the arguments **p** and **q** alias, so we must assume they alias as a worst-case scenario.

So, we must assume that the instruction ***p = 2** affects the data read by **return *q** and thus the latter is data-dependent on the former.

```
int foo(int *p, int *q) {
  *p = 2;
  return *q;
}
```

Figure 2.6: Indirect Data Dependence

2.3.2 Program Dependence Graph

A Dependence Graph is a directed graph in which a vertex A is connected to B when one depends on the other. If A depends on B, it is a matter of convention whether A points to B or vice versa. In this thesis, we will adopt the convention that B points to A.

In a Program Dependence Graph (PDG) [1], the vertices are entities in a program and for the purposes of this thesis, they are LLVM IR instructions. The edges denote control and data dependences. A PDG comes in handy because it captures all the possible dependences in a program and this is why we use it in the following sections.

3. LOOP DISTRIBUTION

We will now turn our attention to the implementation details of an API that illustrates the benefits, and challenges, of decoupling transformations from the rest of the machinery. This strictly transformation-only API is simple enough for the reader to understand it as an example of a broader idea, yet powerful enough to be almost ready for production.

We will consider loop distribution for the simplest and most common case, where we want to distribute the original loop in 2 loops. It is easy to extend both the conceptual idea and the implementation to be able to distribute a loop in more than two loops.

In this simple case, loop distribution essentially consists of extracting some instructions from the original loop and placing them in a new one, which is a copy of the original. More precisely, assume that the set L has all the instructions of the original loop. We want to extract a set $X \subseteq L$ to a new loop. The steps that we will follow are:

1. Create a copy of the original loop
2. Remove the instructions X from the original loop
3. Remove the instructions $L - X$ from the new loop.

Let's now see how we can bring this beautiful high-level idea to an actual implementation and how we can even extend it.

3.1 The Bare-bones of the Transformation

To implement a very basic loop distribution, we need exactly three functions, one for each step outlined above. Their prototypes can be seen in Fig.3.1.

```
Loop *cloneLoop(Loop *OrigLoop, ValueToValueMapTy &MapOrigToNew, LoopInfo &LI,
                DominatorTree &DT);
void removeInstructionsFromOriginalLoop(
  const std::set<Instruction *> &InstsToRemove);
void removeInstructionsFromNewLoop(
  const Loop *OrigLoop, const std::set<Instruction *> &InstsExtracted,
  const std::set<const Instruction *> &InstsToClone,
  const ValueToValueMapTy &MapOrigToNew);
```

Figure 3.1: The Three Basic Primitives of Loop Distribution

Let's begin with some conventions when we describe C++ code.

First, entities in LLVM code, as in a lot of C++ code, are represented and passed around as pointers to an object instead of the object itself. For instance, you can see that **cloneLoop()** does not return a **llvm::Loop**, but rather, a pointer to a **llvm::Loop**. For the rest of this discussion, we will omit the "pointer to a" part for the sake of clarity. For instance, we will say that "**cloneLoop()** returns a **llvm::Loop**" instead of "**cloneLoop()** returns a pointer to a **llvm::Loop**".

Second, we consider different copies of the same instruction with the same contents as different objects. For instance, the instruction **add i32 1, 2** might appear in multiple places in an LLVM IR module. Each place is considered a different **llvm::Instruction**, both in the very reality of the LLVM infrastructure (i.e., it keeps a separate **llvm::Instruction** object for each), but also in our explanations in this text.

Third, the reader does not need to know the exact C++ definitions of types. For instance, the reader does not need to know that a **ValueToValueMapTy** is an alias of **ValueMap<const Value *, WeakTrackingVH>** or what a **ValueMap** or **WeakTrackingVH** are, to understand the essence of the code. They just need to know that it maps a **llvm::Value** to a **llvm::Value**. Similarly, we will refer to entities of type **std::set<T>** as just ”sets” and objects of type **llvm::Loop** as just ”loops”, because the fact that we use this specific implementation of a set or a loop is irrelevant. Thus, in general we will omit such details unless we think they are necessary to understand core properties of the code.

3.1.1 Primitive to Clone a Loop

cloneLoop() takes the original loop in **OrigLoop**, clones it, and it returns us the new loop. However, as you can see, there are 3 more arguments passed. **MapOrigToNew** is passed by reference to be filled. It maps the original instructions to their copies in the new loop. This map will be useful later when deleting instructions. The other two types are passed because we want to preserve them.

We should take a moment to explain the term ”preserve”. In LLVM parlance, usually preserving something means ”keeping it up-to-date”. For instance, let’s say that we have a program dealing with a box. A variable **NumBalls** in this program keeps how many balls are present in this box. Let’s say we have a function whose purpose is to do something with the box. Maybe it adds or removes a bunch of stuff from it, including possibly some balls. We may pass a reference to **NumBalls** to this function so that the function can ”preserve” it. For example, if a call to this function adds two balls to the box, we want it to add 2 to **NumBalls**, thereby keeping the information it holds up-to-date.

Returning to our actual program, we want to preserve the information held in **LoopInfo** and **DominatorTree**. **LoopInfo** contains info about what loops exist in a function, their structure etc. When **cloneLoop()** returns, we would like **LoopInfo** to have been updated automatically. Similarly, dominator tree is useful in many different places inside a compiler. Preserving it automatically is desirable.

Before we go any further, we note that **cloneLoop()** provides a fine level of granularity and minimal coupling among the loop distribution steps. The user can execute this one step and completely skip the rest of the steps. Actually, **cloneLoop()** is one of the many interesting examples of re-usability; the user may not even be interested in loop distribution at all, but she can still use one of its primitive steps because for use cases that are potentially completely different.

This is one of the benefits of providing granular and decoupled APIs. If the API is not granular, or it is but the steps are coupled, the user can use it only for the case that the creator had in mind. But the more granular and decoupled it gets, the more the user can arbitrarily pick individual steps from separate APIs and compose them together in a feast of combinatoric madness.

3.1.2 Pre-conditions to Clone a Loop

There are some pre-conditions which have to hold if we want to clone a loop. These are not so much constraints that our API imposes directly, as much as the LLVM's APIs which **cloneLoop()** uses to clone a loop. Our API just inherits them.

To check these constraints, we provide the following two utilities in Fig. 3.2. The first checks that the instructions we want to extract span exactly one loop. If so, it returns us that loop. The second checks some more detailed structural characteristics of the loop which we do not believe are of great interest to the reader. They are LLVM-specific and do not add to the essence of the transformation.

```
Loop *InstsSpanOneLoop(const std::set<Instruction *> &InstsToExtract, const LoopInfo
bool isLoopDistributable(const Loop *L);
```

Figure 3.2: Utilities Checking Basic Pre-conditions for Loop Distribution

3.1.3 Primitive to Remove Instructions from the Original Loop

The next function is **removeInstructionsFromOriginalLoop()**, taking a single argument, the instructions to be removed. To avoid the obvious, let's not explain what it does but what deserves an explanation is that we pass nothing else to this function. One might expect to pass the loop from which the instructions are going to be removed from, but no.

The reason is that an **llvm::Instruction** has access to its parent, and so when deleting it, it can update its parent automatically. A "parent" in LLVM parlance can mean different things in different contexts, but in most cases, saying "X has Y as its parent" means that the entity in which X is directly enclosed in is Y. What is a parent of what is subjective and it is whatever the LLVM developers have agreed upon. For instance, the parent of a **llvm::BasicBlock** is a **llvm::Function**. A **llvm::BasicBlock** is also inside a bigger entity, a **llvm::Module**, but it is not its parent.

The parent of an **llvm::Instruction** is a **llvm::BasicBlock**. If we delete an **llvm::Instruction**, it updates its parent **llvm::BasicBlock**. But what about the **llvm::Loop** which contains the **llvm::Instruction** ? We don't need to update that because a **llvm::Loop** holds references only its **llvm::BasicBlock**s, not the individual instructions.

The preceding reasoning shows the power of hierarchical structure. When doing a local change, like deleting or adding an instruction, we only need to update its direct parent, in this case a basic block. That basic block is contained in a loop, which is contained in a function, which is contained in a module, but we don't need to update any of those. They get the change for free because each of them considers the enclosed structures as indivisible chunks. If we change something inside a chunk, it still looks like the same chunk to its parent.

3.1.4 Deleting Terminator Instructions

However, there is a caveat; what if we order to delete a terminator instruction? This will make the basic block fundamentally different, meaning, it cannot look like the same chunk to its parent. This in turn will make both the loop and the function fundamentally different and thus we should give them a heads-up. Welcome to the first limitation of our API. We can't extract terminator instructions from the loop, which means that we cannot change its fundamental structure, its blueprint. Down the road, we will see that this will make our life easier in other respects too.

It is important to note that this limitation is not particularly important. Most loop distribution implementations, including the upstream version of LLVM, can't extract terminator instructions. It is a good trade-off because it is rarely useful while it complicates the implementation significantly.

3.1.5 Primitive to Remove Instructions from the New Loop

The last primitive operation is the **removeInstructionsFromNewLoop()**. **InstsExtracted** are the instructions extracted from the original loop (usually it is the same set as the **InstsToRemove** in the previous primitive). What we want is to delete all the other instructions, except these! To do that, we also pass the **OrigLoop**. Under the hood, the function just iterates over all instructions and deletes everything that is not in **InstsExtracted**.

However, there is a problem. We don't want to delete them from the original loop, i.e., **OrigLoop**, but rather from the new one. And we don't want to delete the original instructions but their copies. This is where the previously mentioned **MapOrigToNew** comes in handy. As its name suggests, it maps the original instructions to their new copies. When the function deletes instructions, it goes through the original instructions, checks each of them if it is in **InstsExtracted** (which contains original and not new instructions) and if not, it finds their copy using **MapOrigToNew** and deletes their copy.

Let's try to make that clearer with a made-up example, approaching reality, in Fig.3.3. Each instruction has a unique ID, as a comment next to it and we will refer to them using that. Remember that such a setup is not far from reality as each **llvm::Instruction** is referenced using a pointer to it. So, each instruction can be identified using its address in memory, which is unique.

```
// Original Loop
for (int i = 0; i < n; ++i) {
  a = b + c;  // 1
  d = e + f;  // 2
}
```

Figure 3.3: Identified Instructions

Suppose that we want to extract the instruction with ID **1**. We first call **cloneLoop()** to clone the original loop and get the new one, resulting in Fig.3.4. Note that instructions **1** and **3** are *different* **llvm::Instruction**s. The former is what we call the "original" instruction, while the latter is its "copy". The same is true for instructions **2,4**.

cloneLoop() fills **MapOrigToNew** such that **MapOrigToNew[1] = 3** (i.e., the copy of **1** is **3**) and **MapOrigToNew[2] = 4**.

```
// New Loop
for (int i = 0; i < n; ++i) {
  a = b + c;  // 3
  d = e + f;  // 4
}

// Original Loop
for (int i = 0; i < n; ++i) {
  a = b + c;  // 1
  d = e + f;  // 2
}
```

Figure 3.4: Cloned Loop

Then, we construct a set containing only the ID **1** and pass it to a call to **removeInstructionsFromOriginalLoop**. This results in Fig. 3.5.

```
// New Loop
for (int i = 0; i < n; ++i) {
  a = b + c;  // 3
  d = e + f;  // 4
}

// Original Loop
for (int i = 0; i < n; ++i) {
  d = e + f;  // 2
}
```

Figure 3.5: Removed From Original

Finally, we construct the set **InstsExtracted = {1}** and pass it along with the original loop and **MapOrigToNew** to **removeInstructionsFromNewLoop()**. This function enumerates all the instructions in the original loop, starting with **1**. It checks whether it is in **InstsExtracted** and it is, so, it *skips* deleting its copy (because remember, we want to keep the copies of extracted instructions in the new loop). Then, it deals with instruction **2** and checks whether it is in **InstsExtracted**. It is not, so the function looks up its copy in **MapOrigToNew**, which is **4**, and deletes it. The final state of the code can be seen in Fig.3.6.

```
// New Loop
for (int i = 0; i < n; ++i) {
  a = b + c;  // 3
}

// Original Loop
for (int i = 0; i < n; ++i) {
  d = e + f;  // 2
}
```

Figure 3.6: Removed From New

There is one more argument passed in **removeInstructionsFromNewLoop()**, **InstsToClone**, which we did not address. We will return to this argument later, but first, we need to take a step back and understand the implications of extracting instructions from a loop.

3.2 Dependendeces: The Balkers of Loop Transformations

Let's start by remembering why we extract instructions from a loop in the first place. The reason is that we don't want these instructions to appear in this loop. Either they cause a problem, e.g., they create a loop-carried dependence, or they are in some way heterogeneous with the rest of the instructions, e.g., they are the only ones which we want to unroll. So, basically, we want them to be removed somehow.

But of course, we cannot just delete them, otherwise, we cut out a computation which existed in the original program. Unless we prove it is useless (which we do not do as part of loop distribution), then the original and transformed programs do not do the same thing. This wrecks the correctness of our transformation. To avoid this, we just put them in a new loop, so that they still exist somewhere in the program.

But there are two possible problems that we may give rise to if we move some instructions to a new place. Call L the set of all instructions in the loop and call $X \subseteq L$ the set of the instructions that we want to extract. The first problem is that some instructions in $L - X$ might *depend* on some instructions in X. In this case, we cannot just move X to a different place, as this might potentially break the semantics. Consider the snippet in Fig.3.7.

```
for (int i = 0; i < ...; ++i) {
  A[i] = ...;         // S1
  int l2 = A[i+1];   // S2
}
```

Figure 3.7: L-X Depends on X

Let's say that we want to extract the statement S1, which would be X in this case. S2, along with the loop wrapper code, makes up $L - X$. Here, S2 depends on S1, as S2 reads data from locations to which S1 writes. We should focus on the fact that for any location **A[k]**, for some **k**, the loop *first* reads (with S2) from it and *then* writes to it (with S1). For instance, in the *first* iteration (**i == 0**), S2 reads from location **A[1]** in which S1 writes in the *second* iteration.

Essentially, the reads read the "old" data of the array, i.e., whatever data the array had before we enter the loop. We want to preserve that. But if we extract S1 to a new loop above the original (as we do in the transformation; the situation would be different if we put the new loop *below* the original), *all* the writes will happen before any read. So, for example, we will write to **A[1]** before S2 has any chance to read from it. The approach we have taken is that if any instruction outside X depends on any instruction inside X, then we consider the loop distribution illegal and we provide a utility, which we will mention later, that helps the user to check it (although, the user is free to do the loop distribution anyway).

The second problem is that the set of extracted instructions may not be self-sufficient. It might depend on other instructions, call their set Y, in the loop. So, if we move X around, we need to bring along a copy of Y. But Y might itself depend on another set of instructions Z and so on. In such a case, we say that X depends on Y and it also depends *transitively* on Z. Transitively means that it does not depend directly on Z, but it depends on something which depends on Z. And if Z depends on something, X depends transitively on that too.

The takeaway here is that if we want to extract a set X, we need to bring along a copy of all the instructions which X transitively depends on. Even though we do not need or want to extract or copy them. We say that we need to *clone* these instructions.

Returning back to **removeInstructionsFromNewLoop()**, **InstsToClone** are unsurprisingly the instructions which we need to clone. So, the function will remove all instruction copies in the new loop, except those in **InstsExtracted** and **InstsToClone**.

In the context of **removeInstructionsFromNewLoop()**, ”instructions to clone” might not be the best terminology, because this function does not clone any of these instructions. It just does not delete them. However, by referring to them as ”instructions to clone”, we hope to make clear that these instructions appear in both the original and the new loop.

3.3 Discovering Dependences

Up to here, we have built the bare bones of a loop distribution API which can successfully extract instructions into a new loop. The user can execute the steps separately, combine them rather arbitrarily, or even skip some completely. This is a design that is far from the monolithic nature of a loop distribution pass. Not only is there no profitability or legality involved, but also the transformation is not a single huge step.

From now on, we will build upon this API to achieve different levels of granularity (see Section 2.1.1) and provide useful utilities for the canonical cases. However, at the core, the API will still have the same capabilities.

The first utility we will provide is one that enables the user to automatically find all the instructions that the instructions-to-extract depend on. In other words, the transitive dependences of this set. For this purpose, we will assume that we have a Program Dependence Graph (PDG) (see Section 2.3.2) available.

There are two things to note here. First, assuming that the user's code has some kind of PDG available is a reasonable assumption, because PDG is ubiquitous in compiler optimization. But even if a PDG is not available, the user is free to simply not use our utility to find transitive dependences. They could find them on their own on whatever way they desire. Or, they may pass an empty **InstsToClone** set. Or, even if they use our utility, they could augment **InstsToClone** or remove elements from it. The important thing is that whatever is the use case, the API gives full control to the user to adapt the API to their needs.

In our implementation, we use a straightforward PDG type, named... **PDG**, which is an adaptation of NOELLE's [7] PDG. We do not describe the implementation because we believe it is off-topic for this thesis and not particularly complex or interesting. For the same reasons, we do not describe ways in which the reliance on a PDG could be made more flexible (as for instance the user may want our utility to use a PDG but not our PDG).

Having a **PDG** type available, we introduce the utility **findInstsToClone()** in Fig. 3.8.

```
void findInstsToClone(const std::set<Instruction *> &InstsToExtract,
                      const PDG *LoopPDG,
                      std::set<const Instruction *> &InstsToClone);
```

Figure 3.8: Utility To Find Transitive Dependences

This utility takes the **InstsToExtract** and the PDG of the original loop in **LoopPDG**. It simply does a breadth-first-search over all the transitive edges in **LoopPDG** that point to any instruction in **InstsToExtract**.

A second utility helps the user check if any instructions outside of **InstsToExtract** depend on any instructions inside, depicted in Fig.3.9.

```
bool instsDependOnExtracted(const std::set<Instruction *> &InstsToExtract,
                            const Loop *OrigLoop, const PDG *LoopPDG);
```

Figure 3.9: Utility Which Checks If Any Instructions Depend on Those Extracted

3.4 Back-Channel from the Transformation API to the User

By providing these utilities, particularly **findInstsToClone()**, the API opens the door to a camouflaged but fascinating design pattern. The API becomes what Casey Muratori calls [8] a component instead of a layer. Let's try to understand this by first looking at Fig. 3.10.

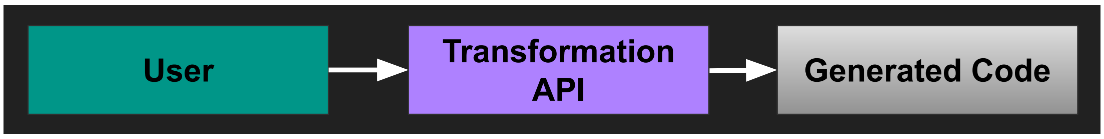

Figure 3.10: Conventional Conceptual Idea of Using a Transformation API

As we have mentioned, in ordinary compiler design, even the idea of separating transformations in their own APIs is revolutionary. But if we do separate them, we think of them sort of what we see in Fig. 3.10. That is, the user decides what they want to transform (e.g., in our case of loop distribution, what instructions should be extracted, what is the original loop etc.) and then calls the API which transforms the code. There is no real communication between the user and the API; the user just orders the API to do something. This is what Casey Muratori calls a layer.

However, a utility like **findInstsToClone()** gives birth to a back-channel of communication, depicted in Fig. 3.11. The user now does not just order the API. It asks the API for information. In this case, "if I want to extract these instructions, what instructions do I need bring along with me?".

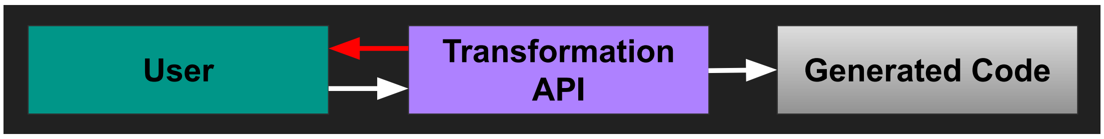

Figure 3.11: Back-Channel Between From the API to the User

Let's see how that becomes useful when the user is a cost-model, as we expect the most common case to be. Suppose that the cost-model tries to decide whether it is profitable to extract some instructions from a loop. It knows the instructions but it does not know on what instructions they depend, i.e., what instructions will be brought along. But it is important to know that in order to cost-model the upcoming extraction correctly!

For instance, maybe the cost-model ponders whether it should extract 3 instructions. And it may cast the extraction profitable. But if it learns that to extract these 3 instructions, it needs to copy along another 40, which may contain slow instructions (like multiple loads and stores), then it might change its mind. Its decision might change because these 40 instructions will now be executed twice, once for each loop. This might cancel out the benefits of removing the 3 instructions from the original loop.

The back-channel essentially provides the user with the ability to see the indirect implications of the transformation that is about to happen; implications that might be difficult to guess (for instance, here the user has to guess the intricate structure of the PDG). What are the options if such a back-channel does not exist?

The first option is for the cost-model to guess. This is not horrible, especially because nowadays cost-models are pretty good at guessing. But it is definitely unnecessary. The cost-model, in this case, should concern itself just with direct decisions like ”should these instructions be removed?”. If we can provide it the implications, we definitely should. Let's not make a cost-model more of a forecaster than it needs to be.

The second option is for the transformation API to take part in the cost-modeling. For instance, the user asks to extract 3 instructions but the transformation API finds out that it needs to copy along another 40. It then takes the liberty to abort the mission in favor of performance. We should appreciate such heroic actions but this one in particular defeats the whole goal of creating a transformation-only API.

3.5 Preserving the PDG

The last utility that we will provide preserves the PDG. We remind the reader that when ”preserve” is thrown around in a compilers-related context, it usually means ”keep up-to-date” and so does here.

Preserving the PDG turned out to be more complicated than expected, so we will devote some space to explain it. The first thing is that we need to have access to alter the full PDG, not just the loop's sub-PDG. ”Full” here depends on the PDG implementation and for ours, it means the PDG for the whole function. The reason we want the full PDG is that adding a new loop means that we transcend the PDG of a single loop (the original).

A second thing to note is that the preservation of the PDG is coupled with the removal of instructions from both loops (essentially, the combination of **removeInstructionsFromOriginalLoop()** and **removeInstructionsFromNewLoop()**). We can trivially break the preservation of the PDG in two steps. One after **cloneLoop()** and one after removing instructions from both loops. What we did not succeed in doing is break the preservation between the two calls that remove instructions. It is essential for the preservation algorithm to assume that the instructions have been removed from both loops. In that regard, our function performs both the preservation and the removal.

With that, we are ready to see the function prototype in Fig.3.12.

```
void preservePDGAndRemoveInstructions(PDG *FunctionPDG, Loop *OrigLoop,
    Loop *NewLoop, const std::set<Instruction *> &InstsExtracted,
    const std::set<const Instruction *> &InstsToClone,
    ValueToValueMapTy &MapOrigToNew);
```

Figure 3.12: Function that Preserves the PDG

Now, let's explain the internals of this function. On a high-level, it is just 2 lines, those in Fig. 3.14.

```
preservePDG(ModulePDG, OrigLoop, NewLoop,
            InstsExtracted, InstsToClone, MapOrigToNew);
removeInstructions(OrigLoop, NewLoop, InstsExtracted,
                   InstsToClone, MapOrigToNew);
```

Figure 3.13: Implementation of preservePDGAndRemoveInstructions

removeInstructions is literally a call to **removeInstructionsFromOriginalLoop()** followed by a call to **removeInstructionsFromNewLoop()**. The meat of the preservation happens in **preservePDG()**. But before we move to its implementation, we wanted to bring to your attention that we preserve the PDG *first* and *then* we remove the instructions. This is because removing the instructions means de-allocating their memory, making them disappear for their parent basic block etc. It just makes our life easier to first manipulate the graph having all the instructions present in memory and on their places, albeit assuming their upcoming deletion, and then actually deleting them.

Before we move to the core of the preservation algorithm, we will present a handy predicate used throughout the code, **IsInNewLoop()**, in Fig.

```
auto IsInNewLoop = [&InstsExtracted, &InstsToClone](const Value *V) -> bool
{
  Instruction *I = (Instruction *)dyn_cast<Instruction>(V);
  if (!I)
    return false;
  return InstsExtracted.count(I) || InstsToClone.count(I);
};
```

Figure 3.14: Check If a Value Will Stay in the New Loop

It's a C++ lambda, which we can think of as a simple function, and it tells us if **V** is an instruction of those that will stay in the new loop. And as we know, those that will stay are the union of **InstsExtracted** and **InstsToClone**.

Let's take a moment to explain some utilities used. The call to the method **count(X)** of a **std::set** counts the number of occurrences of the element **X** in the set. Here, it is used to simply check if an element exists (in which case, **count()** will return a positive result, which will be cast to **true**).

The second utility is **dyn_cast**. It is useful when we want to check if the dynamic type of an element is actually more constrained than its static type. And if so, we cast it to that more constrained type. For instance, **llvm::Instruction** inherits from **llvm::Value**. **llvm::Value** is the more generic type (i.e., an **llvm::Instruction** is always a **llvm::Value**) while **llvm::Instruction** is the more constrained type. A **llvm::Value** is not necessarily an **llvm::Instruction** and that is what we check using **dyn_cast**. If it is, it gives us a **llvm::Instruction** which enables us to use all the auxiliary methods (not available in a **llvm::Value**) of this child type. If not, it just returns **null**.

Now, let's move to the algorithm, which we will split in 4 main parts. There is something important to remember throughout our discussion of this algorithm. We proceed in the algorithm based on the *original* instructions. And we also remind that **InstsExtracted** and **InstsToClone** contain the *original* instructions. Whenever we want to refer to the copy of an original instruction in the new loop, we use **MapOrigToNew**.

The first part of the algorithm is the easiest and it just creates a copy of the sub-PDG of the original loop. We present a simplified version in Fig. 3.15.

```
for (Instruction *I : OrigLoop) {
  if (isInNewLoop(I)) {
    FunctionPDG->createNode(I);
  }
}
```

Figure 3.15: Create a Copy of the Sub-PDG of the Original Loop

The **createNode()** method of a **PDG** wraps an **llvm::Instruction** in a PDG node and adds it to the PDG.

The second and third parts are both two **for**-loops and they live together under the same loop, shown in Fig. 3.16. The second part adds the incoming edges and it is the loop in lines **9-30**. The third part adds outgoing edges and it is the loop in lines **33-47**.

```
for (Instruction *OrigInst : OrigLoop) {
  if (!IsInNewLoop(&OrigInst))
    continue;
  // Get the corresponding new instruction
  Instruction *NewInst = MapOrigToNew[&OrigInst];
  // Get the original node
  PDGNode *OrigNode = FunctionPDG->fetchNode(OrigInst);
  // Add incoming edges to the new node.
  for (PDGEdge *Edge : OrigNode->getIncomingEdges()) {
    // i.e. the tail of the original edge
    const Value *OrigFromValue = Edge->getOutgoingValue();
    // i.e. the tail of the new edge
    const Value *NewFromValue;
    if (!MapOrigToNew.count(OrigFromValue)) {
      // If the original "from" value is _not_ in MapOrigToNew,
      // then it is outside (and is also before) both loops
      // and the NewFromValue is the same as the OrigFromValue.
      NewFromValue = OrigFromValue;
    } else {
      NewFromValue = MapOrigToNew[OrigFromValue];
    }
    FunctionPDG->addEdge(NewFromValue, NewInst);
  }

  // Add outgoing edges to the new node
  for (PDGEdge *Edge : OrigNode->getOutgoingEdges()) {
    const Value *OrigToValue = Edge->getIncomingValue();
    const Value *NewToValue;
    if (!MapOrigToNew.count(OrigToValue)) {
      NewToValue = OrigToValue;
    } else {  // Inside the loop
      NewToValue = MapOrigToNew[OrigToValue];
    }
    FunctionPDG->addEdge(NewInst, NewToValue);
  }
}
```

Figure 3.16: Adding Incoming and Outgoing Edges

We won't describe every part of the code in detail because we believe that it is going to be less effective than if the reader studies the code. However, we will point out some delicate reasoning. For the rest of this discussion, we remind to the reader that in our PDG implementation, if **A** points to **B**, then **B** depends on **A** (and not vice versa).

The one detail has to do with incoming edges to an extracted instruction ("extracted" instructions are also referred as "new" interchangeably), i.e., with the values it depends on. The first case is that the instruction depends on a value outside the original loop, like in Fig. 3.17

```
1 int b = 2;
2 for (int i = 0; i < n; ++i) {
3   a[i] = b;
4 }
```

Figure 3.17: Instruction Depends on Outside Value

Here, the statement in line **3** depends on **b**.

In such a case, where an extracted instruction **A** depends on an outside instruction **X**, there are two things to keep in mind. First, **X** will not be in **MapOrigToNew**. Second, the copy of **A** instruction will depend on **X** too.

The other case is when **X** is inside the original loop. In such a case, the first thing to note is that **X** will be copied along and thus will be in **InstsToClone** because we find the transitive dependences of all the instructions we extract. Unless **X** is also one of the instructions we extract, in which case it will be in **InstsExtracted**. In any case, the copy of **X** will not be removed from the new loop and thus we can create an edge from this copy to our extracted instruction **A**.

Adding outgoing edges follows a similar reasoning.

The final part just removes the nodes from the PDG that correspond to the original extracted instructions. A simplified view is shown in Fig. 3.18.

```
for (Instruction *InstExtracted : InstsExtracted) {
  FunctionPDG->removeNode(InstExtracted);
}
```

Figure 3.18: Remove Nodes for Extracted Instructions

3.6 Ascending the Granularity Ladder

We can provide 2 small utilities which help the user in the canonical cases. For instance, it will be common that the user wants to extract some instructions without needing the control of doing all the steps separately. For this reason, we provide **splitLoopUnchecked()** in Fig. 3.19.

```
void splitLoopUnchecked(const std::set<Instruction *> &InstsExtracted,
                        const std::set<const Instruction *> &InstsToClone,
                        PDG *ModulePDG, LoopInfo &LI, DominatorTree &DT,
                        Loop *OrigLoop);
```

Figure 3.19: Coarse-Grained Call to Split a Loop

This utility automatically clones a loop, removes instructions from both of the loops and preserves the PDG, the dominator tree and the loop info. It is called ”unchecked” in the sense that we assume that the user has checked the pre-conditions in Section 3.1.2. This is why it returns **void**, because it is not supposed to fail. And finally, we also give them the ability to define themselves the instructions to clone.

However, even that function might be too detailed when you just want to split a loop. For this reason, we provide the final and most coarse-grained API function, **splitLoop()**, in Fig. 3.20.

```
bool splitLoop(const std::set<Instruction *> &InstsExtracted,
               PDG *ModulePDG, LoopInfo &LI,
               DominatorTree &DT);
```

Figure 3.20: High-Level Function to Split a Loop

This function gets the instructions to extract, the PDG, the loop info and the dominator tree. It just does all the work mentioned thus far, under the hood. And with that, we believe we have successfully and gradually ascended the granularity of the API, giving the user enough control for the edge cases and enough convenience for the canonical cases.

3.7 Limitations

3.7.1 Removing Terminator Instructions

As we have mentioned in Section 3.1.4, one limitation of this API is that we cannot extract branches out of a loop. This is not so much a weakness on what the API can achieve, because the user is rarely interested to extract branches. But it is a weakness in the usage pattern it enforces. It is an assumption that exists throughout the API but it is not in any way apparent to the user. The user just needs to know that and be sure that all the terminator instructions are cloned, by putting them in **InstsToClone**.

We could have chosen to just assume it explicitly inside the code (i.e., explicitly skip terminator instructions) but this problematic if the user wants to use the low-level primitives. In that case, they may want to delete terminator instructions, which will not cause a problem to the API, and deal with it in any way they like. So, in the end, we think that the current design is a good trade-off which does not withhold control in the extreme cases.

3.7.2 Distributing in More than Two Loops Directly

This API currently lets the user only split a loop in two. Loop distribution is more general in that a loop can be split to an arbitrary number of loops. This can be accomplished with this API. For instance, if we want to distribute in 3 loops, we first split the first loop out, and then we re-split the original loop, to get a total of 3. It is just that the user cannot do it directly, which we believe is not important for a prototype implementation.

4. EVALUATION

4.1 Against API Design Principles

In this section, we will try to evaluate our API against the principles presented in Section 2.1. We do not know of a scientific way to measure the conformance to these principles, because designing based on them is for the most part a trade. However, we do believe that we should take a moment to articulate how we tried to design the API against the trade-offs of these principles.

4.1.1 Granularity

We think that granularity has been explored to its limits! The user can perform the individual steps of cloning, removing instructions and preserving, all the way to just a simple and single call to **splitLoop()**.

4.1.2 Redundancy

We do get a lot of redundancy because of the granularity of the API, however, there is a place where could have provided redundancy in the data types we accept. Currently, we only accept **std::set** for sets and no other type.

There is an infinite number of set implementations so cannot account for all of them in a simple manner. There are alternatives using the C++ type system, but that would probably cause more harm (and complexity) than good. It would also make the API less orthogonal without a tangible benefit. A sensible balance would be attained if we accepted LLVM's set implementation, **DenseSet**, since this code is supposed to live under LLVM infrastructure. We consider this one of the weaknesses of the API against the principles.

4.1.3 Retention

This API retains nothing (such APIs are called stateless or functional)! But we could use some retention. For instance, instead of **InstsExtracted** and **InstsCloned** being passed around continuously, we could retain them.

However, we believe that if the user wants to use the low-level API, it is better give them as much control as possible and not retain anything. The need for passing these things around is a by-product of using the find-grained primitives. If the user does not want to deal with that, then they probably do not need to use the fine-grained primitives anyway. So, they can avoid passing things around by just calling **splitLoopUnchecked()** or **split-Loop()**.

4.1.4 Flow-Control

This API never calls the user back.

4.1.5 Coupling

We consider the API as non-coupled in general. The user can separate the steps without one implying others. However, we will mention again one place where coupling exists and that is the preservation of the PDG which requires the removal of instructions. For more information, see Section 3.5.

4.2 Against Test Cases

In the common case, compiler transformations used during optimization are evaluated by measuring the change in some kind of metric in the final, output code. Usually, this metric is either the running time of the executable or its size. And we evaluate a large and indicative set of input programs trying to discern a statistical improvement in this metric, caused by the transformation.

However, this is not the correct way to evaluate a transformation-only (!) API. First, it is not always legal to apply a transformation even if we can. And second, it is not always profitable to apply a transformation, even if we can. So, to measure for example, the running time improvement that loop distribution might give us, we have to have a legality analysis and profitability heuristic for it. But even if we have them, this measurement will not measure the transformation exclusively. The capabilities of the legality and profitability analyses will influence the measurements.

For this reason, the correct way to evaluate a transformation-only API is to explore the limits of what it can handle, regardless of metrics in the generated code. The evaluation should give us answers in questions like ”Can it remove instructions from any part of the loop? Can it handle all the types of instructions?” etc. And most importantly, whether it can handle the most common cases.

So, we created custom test cases that show the capabilities of this loop distribution API. Before we present the test cases and discuss the API’s behavior over them, we have to take some time to explain the setup and the format of the test cases.

First of all, we have mentioned already, anything presented in this thesis is not LLVM-specific, even though the implementation is over LLVM. For this reason, any code snippets we have seen are in C++ and not in LLVM IR, because one could implement the same API in different compilers with only slight variations. This trend will continue in the test cases, which will be shown in C++. However, we note that the actual test cases were first translated to LLVM IR and then fed to the API.

The second thing we should mention is the way the API was called. The standard work flow in this API starts by a module, which exists inside the compiler, which finds some instructions it wants to extract out of a loop. Then, it calls the API to do this extraction.

We, the human, will pretend to be such a module. Looking at some LLVM IR, we can find instructions we want to extract. But the problem is that we are not inside the compiler. So, we want to design a module that will be, whose only purpose is to act on our behalf and call the API with the instructions we want. The way we do that is simple. First, we stub the instructions we want to extract with metadata, which are data about data.

It might be difficult to wrap your head around the idea of ”data about data”, so we will provide an analogy. Suppose you are filling a form which asks you for your name, phone number etc. Under the box which is to be filled with your phone number, there is another box which asks ”Is this your primary phone number?”, which is followed by two check boxes, ”yes” and ”no”. The first box, asks you about some data; your phone number. The second box, asks you about some data too, but this is data about some *other data* in the form. Hence, it is data about data a.k.a. meta-data.

Returning back to our test cases, having stubbed the instructions we want to extract with metadata, then a module inside the compiler can search for them. It gathers them all, does a bunch of preliminary checks (e.g., that they are all in the same loop) and then calls the loop distribution API. In that way, we have created a work flow where we can stub instructions in LLVM IR and then call the API to extract them. Next, we will present the test cases, by mentioning which parts of the code we stubbed in the LLVM IR.

4.2.1 Simple

```
void foo(int64_t *p, int64_t len) {
  int64_t i = 0;
  int64_t j = 0;
  while (i < len) {
    ++i;
    ++j;  // Stubbed
    p[i] = i;
  }
}
```

Figure 4.1: Simple: Before

Here we simply want to remove the handful of instructions that implement the **++j**. It is a very simple test case and after we pass this through the API, we see Fig.4.2.

```
void foo(int64_t *p, int64_t len) {
  int64_t i = 0;
  int64_t j = 0;
  while (i < len) {
    ++j;
  }
  while (i < len) {
    ++i;
    p[i] = i;
  }
}
```

Figure 4.2: Simple: After

4.2.2 Basic Reordering

```
1 for (i = 0; i < n; i++) {
2   d = D[i];
3   A[i + 1] = A[i] * B[i];  // Stubbed (this whole line)
4   C[i] = d * E[i];
5 }
```

Figure 4.3: Basic Reordering: Before

This is a more realistic test case as we remove line **3**, which creates a loop-carried dependence (see Chapter 1). Besides being a real-world test case, it is also particularly interesting because LLVM's loop distribution cannot handle this. The reason is the legality analysis it uses, which we remind that it is coupled with the transformation. This analysis thinks that to distribute the loop, one would need to reorder lines **2** and **3** and since this analysis does not allow reordering memory operations, this fails.

This failure is yet another example where coupling creates problems. Here, we could simply use a different legality analysis and use the transformation. Or in any case, to put it more simply, the transformation code has nothing to do with the legality-related stuff we described previously. For us, the after can be seen in Fig.4.4.

```
for (i = 0; i < n; i++) {
  A[i + 1] = A[i] * B[i];
}
for (i = 0; i < n; i++) {
  d = D[i];
  C[i] = d * E[i];
}
```

Figure 4.4: Basic Reordering: After

4.2.3 Simple Control-Flow Inside the Loop

```
1 for (i = 0; i < x; i++) {
2   C[i] = D[i] * E[i];
3   A[i + 1] = A[i] * B[i];  // Stubbed
4   if (F[i])
5     G[i] = H[i] * J[i];
6 }
```

Figure 4.5: Simple Control-Flow: Before

In this test case, there is control-flow inside the loop, the **if** in line **4**. As we have mentioned in Section 3.1.4, we are not expected to remove branches using this API, so after loop distribution, the code looks like Fig. 4.6.

```
for (i = 0; i < x; i++) {
  A[i + 1] = A[i] * B[i];
  if (F[i])
}
for (i = 0; i < x; i++) {
  C[i] = D[i] * E[i];
  if (F[i])
    G[i] = H[i] * J[i];
}
```

Figure 4.6: Simple Control-Flow: After

There are a couple of things to note here. First, the stubbed loop-carried dependence was correctly distributed, as we desired. Second, as we expected, the branch in line **4** (in Fig. 4.5) was cloned, because we clone all branches. Third, this branch was not copied alone (i.e., we didn't just clone an empty **if()**), rather everything it was data-dependent on, in this case, **F[i]**, was cloned too, which is another instance that shows that the discovery of dependences works correctly.

However, the body of the **if** is (correctly) not cloned and what was left from the **if**, which is just reading from **F[i]**, has no observable behavior. Thus, we can just delete the branch altogether.

This API will not do that and we would not like it to do it. This is logic that we think is the job of different passes (e.g., passes that do dead-code elimination and Control-Flow-Graph simplification) and these passes will in fact delete the **if**. So, in the end, we will be left with a much nicer code, shown in Fig. 4.7.

```
for (i = 0; i < x; i++) {
  A[i + 1] = A[i] * B[i];
}
for (i = 0; i < x; i++) {
  C[i] = D[i] * E[i];
  if (F[i])
    G[i] = H[i] * J[i];
}
```

Figure 4.7: Simple Control-Flow: After Cleanup

4.2.4 Distributing an Inner Loop

```
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
    C[i] = D[i] * E[i];
    A[j + 1] = A[j] * B[j];  // Stubbed
  }
}
```

Figure 4.8: Distributing Inner Loop: Before

Here we want to test that we can distribute a loop that is inside a loop (such loops are called inner loops). What we want, and get, is shown in Fig. 4.9.

```
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
    A[j + 1] = A[j] * B[j];
  }
  for (int64_t j = 0; j < m; ++j) {
    C[i] = D[i] * E[i];
  }
}
```

Figure 4.9: Distributing Inner Loop: After

4.2.5 Distributing an Outer Loop

```
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
    C[i] = D[i] * E[i];
  }
  A[i + 1] = A[i] * B[i];  // Stubbed
}
```

Figure 4.10: Distributing Outer Loop: Before

Here, we want to test that we can distribute an outer loop, that is, a loop which contains a loop. Similar to the test in Section 4.2.3, the loop distribution will bring all the control-flow with it, so after it the code will look like in Fig. 4.11.

```
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
  }
  A[i + 1] = A[i] * B[i];
}
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
    C[i] = D[i] * E[i];
  }
}
```

Figure 4.11: Distributing Outer Loop: After

But again as in Section 4.2.3, other clean-up passes will us with Fig. 4.11.

```
for (int64_t i = 0; i < n; i++) {
  A[i + 1] = A[i] * B[i];
}
for (int64_t i = 0; i < n; i++) {
  for (int64_t j = 0; j < m; ++j) {
    C[i] = D[i] * E[i];
  }
}
```

Figure 4.12: Distributing Outer Loop: After Cleanup

5. CONCLUSIONS AND FUTURE WORK

The architecture of modern optimizing compilers is not in fact that modern and could be characterized as archaic at this point. We should start being bold about whether fundamental principles of compiler design and their implications are actually useful. For example, the pass-oriented design seems to entice developers to mesh profitability, legality and transformation together. But they are dissimilar, even though they work together towards a common goal, and this entanglement causes problems if people want to manipulate them separately.

In this thesis we have provided an example of how to design a transformation-only API, which can appear as a separate piece of the puzzle of compiler optimization. Moreover, we tried to present how to design APIs which give the user both an easy interaction for the canonical cases and high levels of control for the edge cases.

In the future, we hope to combine such decoupled transformation APIs with powerful profitability analyses to get an end-to-end improvement in compiler optimization. We also believe that well-designed APIs offering high levels of control will not only become useful to the compiler developer but the compiler user too. We aim to expose the compiler internals to the user so that they can affect the compilation pipeline in ever more imaginable ways.

BIBLIOGRAPHY

[1] W. Baxter and H. R. Bauer. The program dependence graph and vectorization. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, page 1–11, New York, NY, USA, 1989. Association for Computing Machinery.

[2] Lawrence Crowl. Sized deallocation - c++ working group paper. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3536.html`, 2013.

[3] LLVM Documentation. Llvm ir. `https://llvm.org/docs/LangRef.html`, 2021.

[4] LLVM Documentation. Loop terminology. `https://llvm.org/docs/LoopTerminology.html`, 2021.

[5] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[6] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[7] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. NOELLE offers empowering LLVM extensions. *CoRR*, abs/2102.05081, 2021.

[8] Casey Muratori. Designing and evaluating reusable components. `https://www.youtube.com/watch?v=ZQ5_u8Lgvyk`, 2004.