Fine-grain characterization of LLVM single passes in a cartography context

M1 Internship report

Avril de Goër de Herve, under supervision by Laure Gonnord (UCBL & LIP),
Co-supervision by Sébastien Mosser and Jean Privat (UQÀM)

Contents

1 Introduction

Working on a compiler to design, implement and evolve static analysis functions or optimizations on intermediate representations is a very tedious task. As an example, the LLVM compiler project, the state-of-the-art research compiler, is more than 10 millions of lines of code, including 2 millions of C++ code[1] for the core library, and 17% of these lines are devoted to the code analysis and optimization part developed as a system of *individual* piece of code names *passes*, orchestrated by a *pass manager*.

Despite the now maturity of C compilers nevertheless, there barely exist software maintenance tools devoted to this particular domain. Of course, pass designers can apply regular software maintenance tools to compilers (*e.g.* to reverse engineer an architecture), but the specificity of compilers is not taken into account. As a consequence, asking simple and recurrent questions such as:

- What is the impact of a language evolution on the compiler's existing passes?
- Where is there any room for pass improvement?
- Where do I insert my new analysis pass in the compilation chain?
- What is the impact in terms of performance of a given pass?

is more difficult than we may think at a first place. The partial documentation, the distributed domain knowledge, and the long lifetime language and compiler software are among the causes of the difficulty. That's why we propose to combine software engineering techniques and compiler knowledge.

This internship takes place in the context of the CAPESA Project[2], whose global objective is to characterise software evolution in terms of "semantic diffs." The CAPESA project proposal identifies two case studies of interest, namely `git` commits and `LLVM` code transformation (*realized by passes*). In the latter, it identifies as a first step the following research challenges:

- **Charting**: How to collect information from the compiler infrastructure, what is the relevant information, is it static or dynamic and how to visualize this information?
- **Understanding**: How to exploit this information as a language engineer to understand better how code evolves?

Regarding LLVM, Sébastien Michelland already worked on these two questions in a previous internship [9]. He developed a number of tools, especially a graph database of interactions between passes in the LLVM compiler, which we can easily manipulate using Neo4j (further described in section 2.2), and a number of tools aimed at facilitating performance analysis were developed in summer 2019 during his internship and Julian Bruyat's lab project [1]. However, the database he produced is based entirely on formal information given by LLVM developers about syntactic interactions between passes and lacks experimental information such as the ability of an optimization to improve the results produced by another, syntactic independent optimization.

Following from his work, we propose a way to characterise pass impact with *pass neutralization*, a method allowing us to produce such dynamic experimental information and in turn broaden our understanding of how passes interact and refine our cartography of the LLVM compiler.

Summary In Section 2, we expose the technical context of our work. Then we focus on our proposition: study the impact of a given pass by *neutralizing* it. In Section 3, we describe the chosen methodology to perform such a neutralization on the `LLVM` 10 infrastructure, on *analyses* as well as

[1] Numbers obtained from the github project, August 2020.
[2] Associated Inria Team, 3 years project, funded 30k euros.

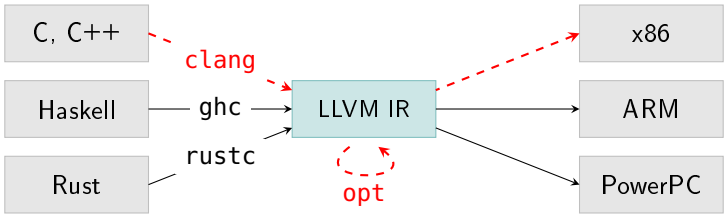

Figure 1: The LLVM compiler infrastructure *Dashed arrows represent the subpart studied in this report*

optimization passes. In Section 4, we perform "compiler-expert driven case studies" in order to experimentally demonstrate the pertinence of the method for pass developers. Finally, we quickly recall some related works in Section 5[3].

2 Background: LLVM and the Neo4j map

2.1 The LLVM Ecosystem

LLVM is the full name of a production compiler that started as the *Low-Level Virtual Machine*[4], a research project at the *University of Illinois* by Lattner and Adve [6] in 2003. This project has contributed to a collection of a modular and reusable compiler and toolchain technologies. It defines an intermediate language-independent code representation, which is based on the *Static Single Assignment* form [4]. LLVM handles high level languages (*e.g.* C/C++, Objective-C, Java) and is able to analyze, transform and optimize arbitrary programs both at compile and runtime. With a lifetime of 17 years and a code base reaching 11 millions of lines of code[5] (LOC) written by more than 1,500 committers, LLVM can be considered as a legacy language infrastructure. It is widely used in research projects and industrial production (*e.g.* its front-end clang is the default compiler for macOS[6]).

In this internship, we restrict our study to the LLVM language infrastructure subset that compiles C and C++ into x86 (see Figure 1). In LLVM, the optimizer opt stands alone as a middle-end between the front and the back-end and is responsible for analyzing and optimizing LLVM's *Intermediate Representation* (IR): inputs and outputs of opt are both LLVM-IR representation format, which facilitates the design of new intermediate passes.

We focus on the part of the source code devoted to the functionalities of passes and their management. opt uses a *pass manager* (further described in section 3.2) which is responsible for orchestrating passes (for instance, it is responsible for scheduling passes in the -O3 meta-optimization). New passes must be registered, and declare their static interactions inside their source code (their dependencies, and the passes they invalidate), so that the pass manager can manipulate them.

The LLVM compiler comes with a test infrastructure, with more than 1k unit tests for Analyses, Optimization and Code Generation[7]; as well as performance tests commonly refered as the *test-suite*. Citing the documentation, "the suite comes with tools to collect metrics such as benchmark runtime, compilation time and code size." This infrastructure thus makes it possible to evaluate a *fixed configuration* on a large set of programs; however it barely permits any minor modification of configuration without recompiling nearly the whole infrastructure. As such, the infrastructure lacks flexibility for pass designers and developers who commonly want to evaluate the **effect of their new pass/modification**

[3]We should at this step mention that some of the outputs of the two internships have been used by Sébastien Mosser and Laure Gonnord during the redaction of a conference submission at the "Software Language Engineering" conference, in July. As a side effect, some text of this report have been widely based on the paper submission, especially the related work.

[4]The full name has become irrelevant to the project since then, but the acronym LLVM stayed.

[5]For comparison, the source code of the Linux kernel is ~25 millions LOC

[6]https://clang.llvm.org/get_started.html

[7]See https://releases.llvm.org/10.0.0/docs/TestingGuide.html.

of pass on the rest of the compilation chain.

2.2 Neo4j

For high-level manipulation of the database gathering the information on interactions between passes we want to have query methods and visualization tools. For this we use the graph database management system Neo4j, following on from Sébastien Michelland who already wrote tools to import the whole set of passes names of LLVM [9]. With passes as nodes and typed edges depicting the different kinds of interactions between passes such as (static declared) dependencies. Figure 2 shows the graph trimmed from all relationships except direct dependencies. Interestingly, it makes it visible that all nodes are either isolated or part of a single component revolving around major analyses such as `domtree` and `targetlibinfo`.

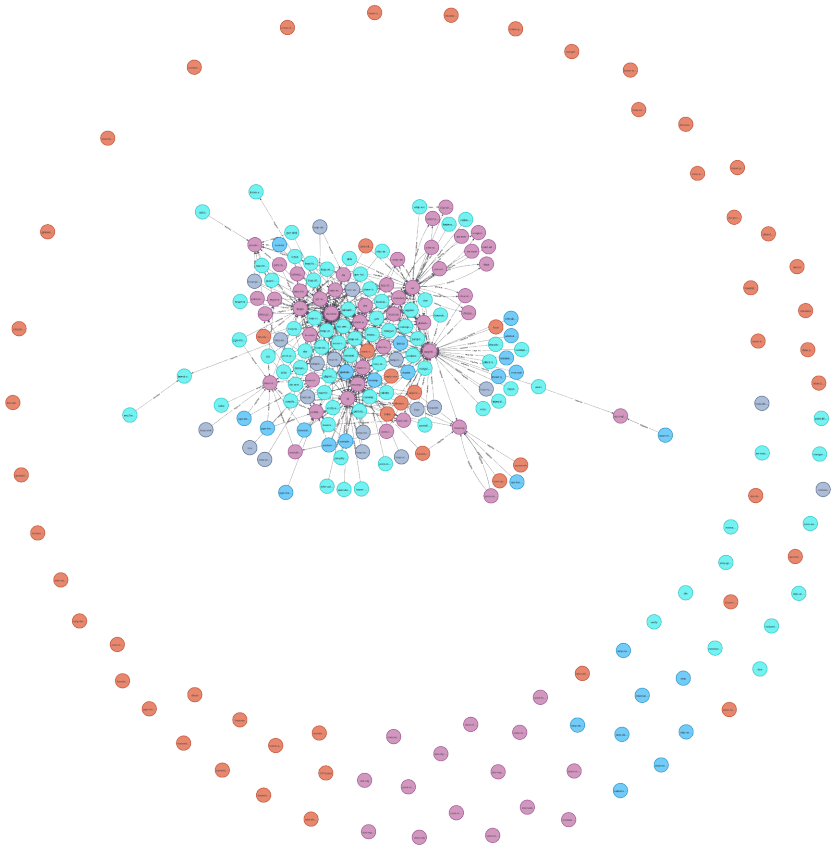

Figure 2: Screenshot showing the graph associated to the database of all passes and direct dependencies. This can be obtained with the minimal query `match (p) return p;` after loading the database into neo4j. Lilac nodes are analyses[8], other colors correspond to various other subclasses of Pass.

Using the `Cypher` query language we can easily query the database, for instance to determine which passes (transitively) depend on the Alias Analysis, as showcased in Figure 3. The query is depicted in Listing 1. Results can be displayed graphically through the use o the "Neo4j browser"[9]

[8]The few analyses at the bottom of the picture which aren't required by any other pass are miscellaneous user-oriented passes such as instruction count and CFG printing.

[9]Contrarily to what the name implies, this web interface can actually be used with a regular web browser rather than as a standalone. It is hosted locally.

Listing 1: Querying the `Neo4j` graph

```
match (p {name: "aa"})<-[:requires*]-(q) return p,q;
```

Our goal is to refine the Neo4j cartography, providing new types of edges and matching additional edges. We call such tools "probes," and in the specific context of this internship we focus on the probe **"impact of a pass on another(s)."**

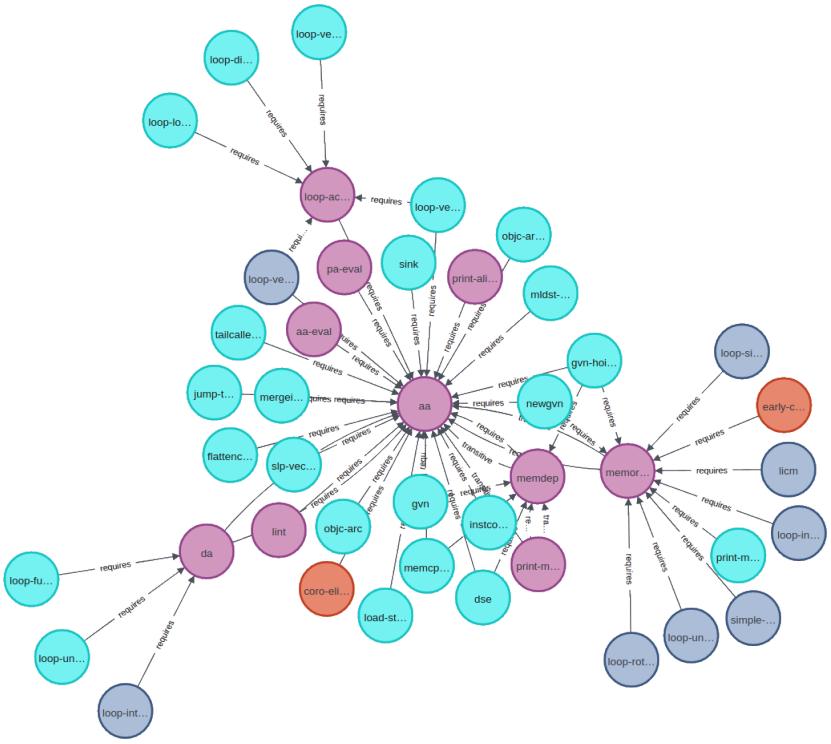

Figure 3: Screenshot of all passes which (transitively) depend on aa (Alias Analysis).

3 Our Proposition: neutralizing passes in LLVM

3.1 Characterizing a single pass: a non-trivial objective

With the goal of providing a precise cartography of the LLVM compiler, we have to study passes at an individual scale and to characterize its interactions with the whole context of the other passes. This is however definitely no simple task.

First, the source code of LLVM is large and complex. Even if passes amount only to 338 thousand lines of code among LLVM's massive 11 millions LOC, many passes amount to thousands LOC on an individual basis (`Scalar Evolution` being no less than 12 thousand LOC). As of LLVM 10.0.0 they are the work of no less than 664 committers[10] over the course of 17 years. Optimizations follow a relatively standard structure[11], but documentation is often incomplete or outdated. As for analyses, they are fully non-standardized, which makes them even harder to study.

[10] Statistics collected in the LLVM GIT repository https://github.com/llvm/llvm-project with the `git-quick-stats` tool, in July 2020.

[11] For instance, `Function Passes` that operate on the granularity of functions *must* implement the `runOnFunction` method, as says https://releases.llvm.org/10.0.0/docs/WritingAnLLVMPass.html#the-functionpass-class.

Second, as mentioned previously, the static information on pass dependencies previously gathered by Sébastien Michelland is only declarative[12] and as such: partial. Indeed, declaration of dependencies and of the preservation of analyses results is declared syntactically by developers, as depicted in Listing 2, in which the `AnalysisUsage.addRequired<>` method is used to declare analyses that are required by a pass, and the `AnalysisUsage.addPreserved<>` method is used to declare analyses that are preserved.

Although understandable for simplicity reasons, this static declaration "of intention" gives however no hint about accuracy. A typical example of this is that the developers only provide information on analyses whose results are guaranteed to be preserved by a given pass. Still, there is no guarantee that an analysis who is not listed as preserved by another pass is indeed invalidated by it.

Listing 2: A portion of the source code of the Loop Invariant Code Motion pass, LLVM10

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addPreserved<DominatorTreeWrapperPass>();
  AU.addPreserved<LoopInfoWrapperPass>();
  AU.addRequired<TargetLibraryInfoWrapperPass>();
  if (EnableMSSALoopDependency) {
    AU.addRequired<MemorySSAWrapperPass>();
    AU.addPreserved<MemorySSAWrapperPass>();
  }
  AU.addRequired<TargetTransformInfoWrapperPass>();
  getLoopAnalysisUsage(AU);
}
```

Furthermore, LLVM passes are complex enough, and programs diverse enough to make this static information completely insufficient, and it is absolutely possible that the results of an analysis declared as a dependency of another be actually totally unused in a given practical case.

Finally, the performance of a compiler is determined by the whole set of passes it uses and their scheduling. It is thus difficult to study the performance of a single pass, since it is fully dependent on the context in which it is used. Thus, performance study of an isolated pass is most likely fully irrelevant. This adds up to the usual difficulty of performance evaluation, which requires to establish a set of metrics that be relevant to study the performance of a given piece of software. In the case of LLVM, this task has to be renewed for each pass, since metrics relevant to the study of one might be fully irrelevant for another.

3.2 Pass manager and the idea of pass neutralization

As previously mentioned, passes in LLVM are invoked by the tool `opt` through the *pass manager*[13], which is fully responsible for scheduling them, calling them, and interfacing them with each other. For instance, when calling `opt -licm`, the pass manager ensures that dependencies of `licm` are all run beforehand. This is naturally convenient from a user perspective, since it is much more practical to call only a few chosen optimizations (or most often: a meta-optimization such as `O3`), rather than an exhaustive list of all necessary optimizations and analyses that take into consideration *all dependencies involved and the data of which passes invalidate others*.

However, this also entails more difficulty when we try to assert the impact of a pass. Indeed, passes in meta-optimizations such as `O3` are dynamically scheduled and might be run several times. This is true not only for analyses, which is expected in case one is required by multiple others and sees its results invalidated by some, but also for optimizations, as illustrated by Listing 6 (in Appendix A.1): analysis `aa` is called 21 times during the execution, while optimization `licm` is called 3 times. For lack of a definite static scheduling, it is thus impossible to study the impact of one of them on the results given by the meta-optimization as a whole by simply removing it from the schedule and running tests against the normal meta-optimization.

[12] https://releases.llvm.org/10.0.0/docs/WritingAnLLVMPass.html#specifying-interactions-between-passes

[13] In concrete terms, the `PassManager` class. See http://llvm.org/docs/WritingAnLLVMPass.html#what-passmanager-does, https://llvm.org/doxygen/classllvm_1_1PassManager.html, and https://llvm.org/doxygen/PassManager_8h_source.html for documentation.

Furthermore, even if it is possible to explicitly run two static sets of optimizations against one another in order to evaluate the impact of adding or removing one of them from the set, this is something we simply cannot do as the pass manager always calls dependencies of a pass even if a user does not invoke them explicitly. As a matter of fact, it would anyway be impossible to try and run an optimization without its dependencies as long as we respect the hypothesis that no passes are registered as dependencies without being actually used by the pass.

For these reasons, we propose studying interactions between passes through the approach of *neutralization*, that is: to directly modify a pass so as to render it inactive. By tweaking the passes themselves, it is possible to nullify the results given by a pass without involving the pass manager. It can thus work normally, which allows us to gather statistics on the results given by a dynamically scheduled optimization group within which some passes are disabled. It also makes it possible to deactivate some analyses and observe the impact this has on an optimization which depends on it. Overall, we believe this offers the most accurate comparison points to the normal functioning of LLVM, and as such, that it is the best way to study the impact of a pass *within its context*.

3.3 Neutralizing a pass

Here we will go into further detail about how we proceed to perform pass neutralization concretely, on the source code of `LLVM`.

The most straightforward approach, naturally, is to simply hollow the source code of a pass completely and to leave only syntaxic statements such as the declaration of required or preserved analyses (as seen in Section 3.1). However, the very principle of analyses is to provide information that can then be engineered by subsequent passes. As such, the source code of an analysis includes the construction of an interface that can be used to communicate with passes that depend on it. It is thus impossible to blindly remove code from its source file, because it would lead related passes to refer to a non-existent data structure, and thus render the whole optimization process non-operational. Instead, we need to proceed carefully and preserve the creation of the data structure, and to disable only the parts of the code that fill it with information.

Furthermore, in a long term approach, a convenient way of deciding what passes to enable and which ones to neutralize is desirable if we are to thoroughly study how passes interact with one another. With this in mind, the destructive approach of simply removing parts of the source code is definitely impractical as it would entail keeping a copy of each pass' source file and the systematic swapping between the two copies (normal and neutralized). Mass commenting code could be an option, but hardly a more practical one.

Thus, we chose the more subtle approach consisting in searching for relevant control points allowing us to fully disable a pass by doing only minor modifications to the code. Ultimately, conditional macros (`ifdef...`) tests could be inserted in the code to decide whether or not to enable a pass, and controlled from a single file using global variables, though in this internship we merely relied on commenting to disable calls to subsequent functions.

Listing 3 presents the control points for the optimization `Dead Store Elimination`, aka `DSE`. The functioning of the pass relies on two master calls to `eliminateDeadStores()`, which recursively calls all other functions that constitute the pass. Disabling this master calls allows us to fully neutralize the pass.

3.4 Toward a standard neutralization protocol for optimizations

The two master calls to `eliminateDeadStores()` in `DSE` are located at very specific points in the code (an extract is displayed in Listing 7, in Appendix A.2). The first is part of a method aiming to communicate analyses that are preserved by the pass (`PreservedAnalyses DSEPass::run`, line 1). As for the second, it belongs to the definition of class `DSELegacyPass` as a subclass of `FunctionPass`

Listing 3: Neutralizing the DSE optimization pass

```
$ diff \
  ./lib/Transforms/Scalar/DeadStoreElimination.cpp \
  ./extras/neutralized-DSE.cpp
1361c1362,1363
<   if (!eliminateDeadStores(F, AA, MD, DT, TLI))
---
> //  Pass neutralization
> //  if (!eliminateDeadStores(F, AA, MD, DT, TLI))
1383c1385,1386
<     if (skipFunction(F))
---
> // Pass neutralization
> // if (skipFunction(F))
1393a1397
>     return false;
```

(line 21), that is, the class of passes that run on each function in the program independently from the others. More specifically, the call occurs in the definition override of the method `runOnFunction()` (line 29).

This second call is part of the standard protocol for writing passes. As described in the documentation of LLVM[14], all optimizations are expected to be written as sub-classes of one among `ModulePass`, `CallGraphSCCPass`, `FunctionPass`, `LoopPass`, or `RegionPass`. Each of those classes has a matching `runOn...` method, which effectively commands the execution of the pass, and it is expected that every optimization's source code contains similar calls to master functions whose disabling would effectively neutralize the pass.

It should however be noted that as mentioned above, the source code of `DSE` includes another call to such a master function, which seems to operate fully independently from that of the `runOnFunction` method, and that the underlying method is *not* described in the documentation. As such, our hypothesis should be considered with caution and we cannot fully assert its correctness—nor can we blindly rely on the documentation to search for control points. Nonetheless, this should be the first place to look when attempting to neutralize a pass, and it should at least be a major control point.

3.5 Neutralizing an analysis pass

In the previous subsection, we explained how some relatively standard structure shared by optimizations allows for a unified neutralization protocol that should work on any optimization. Analyses, however, appear to lack the use of any standard, and are also ill-documented. Therefore, we cannot offer any generic method to neutralize analyses, and instead we have to study each pass at an individual scale, which entails much more work as studying code to understand its structure is anything but a simple task, all the more so as the code of LLVM's analyses is often quite complex and obscure.

Due to this, the results we can offer in this area are much lesser than what we hoped for. In fact, the only result we can offer is the neutralization of `AliasAnalysis`. S. Michelland had already provided this result [9], though his report doesn't describe exactly how. Nonetheless, the hints he gave were enough to allow us to pinpoint the right control point much more easily. Listing 4 shows the little tweaks applied to neutralize the pass. More context about the matching code section can be found in Listing 8, in Appendix A.3.

4 Using pass information: a case study

In this section we study the particular case of the Alias Analysis pass, and how our approach can help the compiler designer to improve this crucial pass inside LLVM.

[14] https://releases.llvm.org/10.0.0/docs/WritingAnLLVMPass.html#pass-classes-and-requirements

Listing 4: Neutralizing the Alias Analysis wrapper pass

```
$ diff llvm/lib/Analysis/AliasAnalysis.cpp neutralized-passes/Analysis/AliasAnalysis.cpp
113,117c113,118
<   for (const auto &AA : AAs) {
<     auto Result = AA->alias(LocA, LocB, AAQI);
<     if (Result != MayAlias)
<       return Result;
<   }
---
> // Pass neutralization
> //  for (const auto &AA : AAs) {
> //    auto Result = AA->alias(LocA, LocB, AAQI);
> //    if (Result != MayAlias)
> //      return Result;
> //  }
```

4.1 Why Alias Analysis?

Alias Analysis[15] is one of the key analysis passes in the LLVM compiler, for which there exists a `Manager`, which is used as a query provider for optimizations passes. All Alias Analyses implementations must follow the specification depicted in Section A.4.

Designing precise Alias Analyses is still an important research topic, because a large number of optimizations rely on them. For instance, in LLVM 10, O3 has 21 instances of calls to aa before optimizations such as `licm` (Loop Invariant Code Motion), and `dse` (Dead Store Elimination).

Like other analyses, it is "quite easy" to demonstrate a precision improvement of a given new alias analysis (because it has been proposed with some particular shapes of programs in mind): a common evaluation strategy for alias analyses is based on the exhaustive static enumeration of all pairs of pointers of a given source file, and the comparison of the total numbers of pairs for which each analysis answers "they do not alias for sure".

However, for the analysis developer, finding a metric to demonstrate the global impact of their analysis on other analyses or optimization passes is more difficult than expected, and Chapter II.4 of [8] was a first attempt toward it. This tedious experimental analysis has demonstrated the need for more general evaluation strategies and was the very first motivation of the global CAPESA project.

4.2 First step : identifying client passes for Alias Analysis

We already showed in Figure 3 how we can query the initial graph to determine which optimizations passes depend on the Alias Analysis. All non-lilac nodes are potential client optimization passes of interest.

For the rest of the study, we chose the `DSE` pass (Dead Store Elimination) among these code optimizations for the following reasons:

- The objective of this optimization is well known folklore[16].
- The number of lines of code (less than 1500) is short enough to have enough confidence on our code neutralization strategy.
- The number of calls to the `Alias Analysis` analysis pass (45 calls to $AA*$ functions inside its source code) is one of the biggest amount of calls to $AA*$ functions in the Scalar Code Optimization directory.
- We think it is technically easier to implement a good performance measure on DSE than LICM (Loop Invariant Code Motion) where we should detect the number of code motions that have been performed).

[15] https://en.wikipedia.org/wiki/Alias_analysis
[16] See for instance https://en.wikipedia.org/wiki/Dead_store.

From this first step, we now have enough confidence to study DSE as an optimization pass in isolation, and then study the impact of AA on DSE.

4.3 Impact of a pass on performance: the example of DSE

To characterize the impact of DSE on the `LLVM` infrastructure, we use our pass neutralisation on the DSE pass, and use as impact metric the dynamic number of instructions effectively launched at runtime.

We run a modified compilation infrastructure for the `LLVM` standardized benchmarks, using tools developped by J. Bruyat last year [1], one time to run the original version of DSE, and a second time to measure the performances of the neutralized version. We obtain a graph depicted in Figure 4, from which we can conclude that there are three classes of programs: a vast majority of programs for which DSE has nearly no effect, and two other classes.

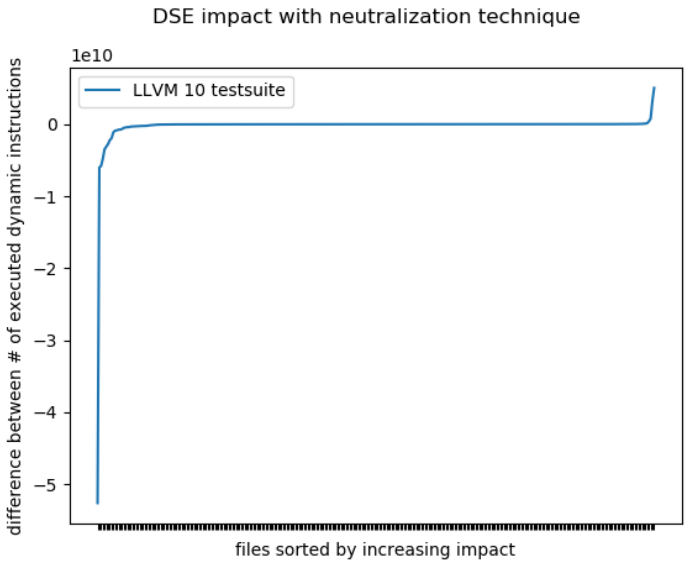

Figure 4: Measuring the impact of DSE (LLVM10)

With the help of Python `statistics` module, we can refine the preceeding result by asking for the 10% most impacted files (10th decile), and thus trying to find a common pattern in these files. The `llvm-diff` tool, that can show differences between intermediate representations, also helps finding interesting parts of code in files. For instance, the source file that is "the most impacted by DSE" is `RawSubsetBbenchmarks.cpp`, which contains many instances of the pattern depicted in Listing 5, for which

Listing 5: A portion of the source code of `RawsubsetB` benchmark, LLVM10

```
for (Index_type i=0 ; i< state.range(0) ; i++ ) {
       out1[i] = in1[i] * in2[i] ;
       out2[i] = in1[i] + in2[i] ;
       out3[i] = in1[i] - in2[i] ;
     }
```

This second step gives us two partial answers/information for our global analysis: DSE is clearly one of the most important code transformations relying on Alias Analysis, and it performs well on array intensive piece of codes, for which we know that Alias Analysis is of crucial importance.

4.4 Refining the previous analysis: impact of AA on DSE

To refine the previous analysis, and thanks to the code neutralization of the Alias Analysis pass, we are now able to generate statistics for the two following combinations: "`aa,dse`" and "`neutralized(aa),dse`".

Figure 5 shows the graph obtained from this statistics. Here again we have three data regions to analyze.

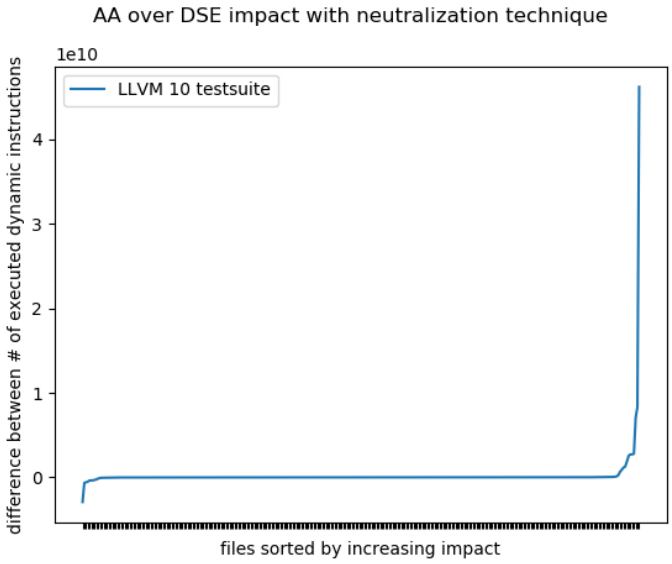

Figure 5: Measuring the impact of AA on DSE (LLVM10)

A quick look at the $10th$ decile show that the `RawSubsetB` "potentiality of optimization by DSE" is successfully detected by the current version of the `Alias Analysis` of LLVM.

Another look at the first decile gives an interesting code `stepanov`, specially designed to evaluate C compilers when adding elements of an array (in 13 different ways, most of them using pointer arithmetic). From this result, we can conclude that there is still a room of improvement of Alias Analysis in `LLVM`, especially when it comes to specifically trigger DSE optimizations inside code that intensively use pointer arithmetic. This goes in the same direction as M. Maalej's PhD statement [8], but in a more systematic way.

In conclusion of this section, we have shown that we are able to use our methodology to guide a pass developer while evaluating their pass evolution/improvement. In future work we plan to import all the information we gained on the passes `aa`, `dse`, `licm` during this study in the LLVM Cartography. These information will refine the information of the dependencies `aa-dse` and `aa-licm` with more dynamic information about the declared dependencies.

5 Related work

This related work is the compilation-oriented part of the SLE submission, that we reproduce here with barely no modification.

Compiler validation A large part of compiler literature has focused on compiler testing, mainly for debugging purposes, we refer the reader to the survey [3] and the empirical study [2] for a comprehensive overview of these methods in production compilers. As for static analyses inside compilers, very few

works have tried to address the evaluation of their precision in a generic way [10]. Validating compilers is still a hot topic, and the complexity of production compilers has increased the need for more trustable tests, especially when it comes to validate aggressive compilation [7].

Reference benchmarks. Some communities have converged towards standard benchmarks for which they informally agreed on dynamic measures of efficacity (timing, efficient acceleration on parallel machines) for method comparisons. For example, nearly all works about polyhedral kernels/programs use Polybench[17] as part of their experimental setting. Still, compiler optimization designers struggle to evaluate the impact of their analyses and optimization. Various communities use this standard benchmark, from the IMPACT specialized workshop[18] to more general-purpose conference [11] in the *International Symposium on Code Generation and Optimization* (CGO) or the *Programming Language Design and Implementation* (PLDI). In these papers, the experimental sections are complex, and the community has evolved to ask for more reproducible results via the evaluation of artifacts[19].

Pass interactions. Focusing on the understanding of passes interactions and the associated optimization, the incremental compiling technique is interesting [5]. Using this technique, a program is compiled in a given way, its performances are measured, and it is then recompiled using another set of passes until the process yields a convenient binary. Other approaches using artificial intelligence were defined, to select the pass scheduling that might fit the best a given program, based on structural information extracted from its source code [12]. The main drawback of these works is that they do not consider the language engineer as a first-class citizen, and do not provide any explanation to her. Thus, if one can use these techniques to optimize a given program, for the language engineer, there is no useful information to support her work.

6 Conclusion

We have proposed a global approach driven by compiler development research questions in order to propose a dynamic approach for measuring impact of passes improvement in the `LLVM` compiler. Pursuing the previous work on LLVM cartography, we propose a refinement of the previous collected information with fine-grain dynamic analyses by *pass neutralization*, for which we propose a general methodology.

Our approach is illustrated in the particular case of the LLVM10 compiler infrastructure. However, we claim that the (theoretical and practical) complexity of measuring the impact of a given pass would be the same for other compilers like GCC or ICC 11.

The experimental and empiric approach we promote might also suffer from uncertainty from the empirical measures point of view, especially when it comes to measuring performance or precision. Improving the accuracy of the measure is a complementary activity, that should be realized by compiler experts.

[17] http://web.cs.ucla.edu/~pouchet/software/polybench/
[18] http://impact.gforge.inria.fr/
[19] https://www.artifact-eval.org/

References

[1] Julian Bruyat. Outillage pour l'étude de l'impact de l'ordre des passes de LLVM. https://laure.gonnord.org/pro/papers/rapport_POM2018_Bruyat.pdf, 2018.

[2] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, pages 180–190, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing methods. *ACM Comput. Surv.*, 53(1), February 2020.

[4] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions On Programming Languages and Systems*, 13(4):451–490, 1991.

[5] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Embedded processor design challenges. pages 171–187, New York, NY, USA, 2002. Springer-Verlag New York, Inc.

[6] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization CGO, 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE, 2004.

[7] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1052–1065, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Maroua Maalej Kammoun. *Low-cost memory analyses for efficient compilers*. Theses, Université de Lyon, September 2017.

[9] Sébastien Michelland. Exploration et cartographie des passes de LLVM. http://perso.ens-lyon.fr/sebastien.michelland/llvm/RapportSeb/rapport.pdf, 2018.

[10] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pages 81–93, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. Locus: A system and a language for program optimization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 217–228. IEEE Press, 2019.

[12] Z. Wang and M. O'Boyle. Machine Learning in Compiler Optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

A Technical Appendix

A.1 Dynamic execution of passes

In Listing 6 we depict the dynamic list of passes called by the pass manager when executing -O3.

Listing 6: List of the passes called by the pass manager during the execution of O3

```
$ opt bar.bc -O3 --debug-pass=Arguments -o /dev/null
Pass Arguments:  -tti -tbaa -scoped-noalias -assumption-cache-tracker -targetlibinfo -verify -ee-instrument -
     simplifycfg -domtree -sroa -early-cse -lower-expect
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -tbaa -scoped-noalias -assumption-cache-tracker -
     profile-summary-info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-
     propagation -attributor -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops -lazy-
     branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-
     eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -memoryssa -early-cse-memssa -
     speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simplifycfg
      -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-
     emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-
     freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-
     remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-
     verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate -memoryssa -licm -loop-unswitch -
     simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
     instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-
     deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq
     -opt-remark-emitter -gvn -phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa
      -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-
     threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -basicaa -aa -memoryssa -loops -
     loop-simplify -lcssa-verification -lcssa -scalar-evolution -licm -postdomtree -adce -simplifycfg -
     domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier
      -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldce -basiccg -globals-aa -domtree -
     float2int -lower-constant-intrinsics -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa
     -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
      -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -loop-accesses -demanded-bits
     -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution
     -aa -loop-accesses -lazy-branch-prob -lazy-block-freq -loop-load-elim -basicaa -aa -lazy-branch-prob -
     lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -domtree -loops -scalar-evolution -basicaa
      -aa -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -slp-vectorizer -opt-remark-
     emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -loop-unroll -lazy-
     branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -memoryssa -loop-simplify -lcssa-
     verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
     transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -constmerge -domtree -
     loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-evolution
     -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-
     pairs -simplifycfg -verify -write-bitcode
Pass Arguments:  -domtree
Pass Arguments:  -targetlibinfo -domtree -loops -branch-prob -block-freq
Pass Arguments:  -targetlibinfo -domtree -loops -branch-prob -block-freq
```

A.2 Source code of Dead Store Elimination

Listing 7: Part of the source code of dse showcasing the two control points for pass neutralization

```
PreservedAnalyses DSEPass::run(Function &F, FunctionAnalysisManager &AM) {\ref{lst:dse-source:preserve}
  AliasAnalysis *AA = &AM.getResult<AAManager>(F);
  DominatorTree *DT = &AM.getResult<DominatorTreeAnalysis>(F);
  MemoryDependenceResults *MD = &AM.getResult<MemoryDependenceAnalysis>(F);
  const TargetLibraryInfo *TLI = &AM.getResult<TargetLibraryAnalysis>(F);

//  Pass neutralization
//  if (!eliminateDeadStores(F, AA, MD, DT, TLI))
    return PreservedAnalyses::all();

  PreservedAnalyses PA;
  PA.preserveSet<CFGAnalyses>();
  PA.preserve<GlobalsAA>();
  PA.preserve<MemoryDependenceAnalysis>();
  return PA;
}

namespace {

// A legacy pass for the legacy pass manager that wraps \c DSEPass.
class DSELegacyPass : public FunctionPass {
public:
  static char ID; // Pass identification, replacement for typeid

  DSELegacyPass() : FunctionPass(ID) {
    initializeDSELegacyPassPass(*PassRegistry::getPassRegistry());
  }

  bool runOnFunction(Function &F) override {
    // Pass neutralization
    // if (skipFunction(F))
      return false;

    DominatorTree *DT = &getAnalysis<DominatorTreeWrapperPass>().getDomTree();
    AliasAnalysis *AA = &getAnalysis<AAResultsWrapperPass>().getAAResults();
    MemoryDependenceResults *MD =
        &getAnalysis<MemoryDependenceWrapperPass>().getMemDep();
    const TargetLibraryInfo *TLI =
        &getAnalysis<TargetLibraryInfoWrapperPass>().getTLI(F);

    return eliminateDeadStores(F, AA, MD, DT, TLI);
    return false;
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesCFG();
    AU.addRequired<DominatorTreeWrapperPass>();
    AU.addRequired<AAResultsWrapperPass>();
    AU.addRequired<MemoryDependenceWrapperPass>();
    AU.addRequired<TargetLibraryInfoWrapperPass>();
    AU.addPreserved<DominatorTreeWrapperPass>();
    AU.addPreserved<GlobalsAAWrapperPass>();
    AU.addPreserved<MemoryDependenceWrapperPass>();
  }
};
```

A.3 Source code of Alias Analysis

Listing 8: Part of the source code of aa showcasing pass neutralization

```
//===----------------------------------------------------------------------===//
// Default chaining methods
//===----------------------------------------------------------------------===//

AliasResult AAResults::alias(const MemoryLocation &LocA,
                             const MemoryLocation &LocB) {
  AAQueryInfo AAQIP;
  return alias(LocA, LocB, AAQIP);
}

AliasResult AAResults::alias(const MemoryLocation &LocA,
                             const MemoryLocation &LocB, AAQueryInfo &AAQI) {
// Pass neutralization
//  for (const auto &AA : AAs) {
//    auto Result = AA->alias(LocA, LocB, AAQI);
//    if (Result != MayAlias)
//      return Result;
//  }
  return MayAlias;
}
```

A.4 LLVM 10.0 Alias Analysis Specifications

In the documentation of LLVM 10.0, reproduced here in a verbatim way, the *Alias Analysis*[20] semantics is defined as the following:

- The `NoAlias` response may be used when there is never an immediate dependence between any memory reference based on one pointer and any memory reference based the other. The most obvious example is when the two pointers point to non-overlapping memory ranges. Another is when the two pointers are only ever used for reading memory. Another is when the memory is freed and reallocated between accesses through one pointer and accesses through the other — in this case, there is a dependence, but it's mediated by the free and reallocation.
- As an exception to this is with the `noalias` keyword; the "irrelevant" dependencies are ignored.
- The `MayAlias` response is used whenever the two pointers might refer to the same object.
- The `PartialAlias` response is used when the two memory objects are known to be overlapping in some way, regardless whether they start at the same address or not.
- The `MustAlias` response may only be returned if the two memory objects are guaranteed to always start at exactly the same location. A MustAlias response does not imply that the pointers compare equal.

[20] https://releases.llvm.org/10.0.0/docs/AliasAnalysis.html