Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Lukáš Krížik

Bobox Runtime Optimization

The Department of Software Engineering

Supervisor of the master thesis:	RNDr. Filip Zavoral, Ph.D.
Study programme:	Informatics
Specialization:	Software Systems

Prague 2014

I would like to thank my family and friends for a persistent support in studies. I would also like to thank the supervisor for the patience and helpful advice.

Název práce: Bobox Runtime Optimization

Autor: Lukáš Krížik

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Filip Zavoral, Ph.D.

Abstrakt: Cílem této diplomové práce je vytvořit nástroj na optimalizaci kódu pro paralelní prostředí Bobox. Nástroj redukuje počet krátce a dlouze běžících úloh na základě statické analýzy kódu. Některé případy krátce běžících úloh způsobují zbytečné přeplánování. Pokud plánovač nemá dostatek informací o dané úloze, plánovač může úlohu naplánovat, i když tato úloha nemá všechna potřebná vstupní data. Pro odstranění krátce běžící úlohy nástroj analyzuje použití vstupních dat a informuje plánovač. Dlouze běžící úlohy můžou v některých případech potlačit paralelismus. Větší granularita úloh může znatelně vylepšit časy běhu v paralelním prostředí. Pro odstranění dlouze běžících úloh nástroj musí být schopen vyhodnotit složitost kódu a vložit příkaz pro přeplánování na vhodné místo.

Klíčová slova: bobox, statická analýza kódu, optimalizace, složitost kódu, clang

Title: Bobox Runtime Optimization

Author: Lukáš Krížik

Department: The Department of Software Engineering

Supervisor: RNDr. Filip Zavoral, Ph.D.

Abstract: The goal of this thesis is to create a tool for an optimization of code for the task-based parallel framework called Bobox. The optimizer tool reduces a number of short and long running tasks based on a static code analysis. Some cases of short-running tasks cause an unnecessary scheduling overhead. The Bobox scheduler can schedule a task even though the task does not have all input data. Unless, the scheduler has enough information not to schedule such task. In order to remove such short-running task, the tool analyses its input usage and informs the scheduler. Long-running tasks inhibit a parallel execution in some cases. A bigger task granularity can significantly improve execution times in a parallel environment. In order to remove a long-running task, the tool has to be able to evaluate a runtime code complexity and yield a task execution in the appropriate place.

Keywords: bobox, static code analysis, optimization, code complexity, clang

Contents

1. Introduction

The Bobox project is a task-based framework for parallel computing. In such a framework, short-running tasks cause bigger CPU consumption by the framework itself and long-running tasks can inhibit the parallel execution. A static code analysis can be used to detect and eliminate such execution paths in the user code. Since the core and interface language of the Bobox framework is C++, static code analysis becomes more difficult in proportion to the lack of tools. This has been the biggest pitfall for any static analysis of C++ code. However, it has become less marginal with a growing support for tooling in the Clang compiler front-end [14]. Clang exposes C++ code as a user-friendly *Abstract Syntax Tree (AST)* [17] structure.

1.1 Goals

Apart from this text, the main asset of this thesis is a tool for optimizing code using the Bobox framework. By analysing AST, the tool is able to diagnose and potentially transform user code to eliminate both short and long execution paths. There is an implementation of the two different patterns of unoptimized usage in the context of this thesis. However, the tool is designed to be easily extensible with new optimization methods. Both implemented optimization methods are able to inject new code to give the Bobox internal facilities information about the user code structure. The tool is implemented using the Clang tooling interface [15]. Therefore, it inherits all the Clang limitations, such as its platform support.

1.2 Structure of the thesis

The second chapter begins with a brief description of the Bobox framework, in order to familiarize a reader with its underlying mechanisms. The reader should understand why the user code can be optimized using the transformations mentioned later in chapters related to specific optimization methods.

The third chapter describes the problems of static code analysis first of all in general, and then specifically for the C++ language since it is the core and interface language of the Bobox framework. The chapter also describes the possible approaches to a static code analysis of C++ code, their advantages and disadvantages, the internal implementation, and the user interface and limitations. The last section in this chapter addresses related work.

Because the chosen approach to implement the optimizer tool was the Clang tooling interface, Chapter 4 is dedicated to its detailed description. Tools implemented on top of the Clang front-end are equipped with an abstract syntax tree representation of code. One section describes the design of this data structure, and the possibilities of its traversal. Possibilities of source-to-source transformations are described in the following section. The Clang front-end itself provides multiple interfaces for an implementation of static code analysis tools.

The next two chapters introduce implemented optimization methods. Each chapter contains sections with information related to the Bobox framework, a detailed description of the algorithms used to detect an unoptimized usage of the framework and other details of the implementation.

Chapter 7 offers a high-level look at the tool design and some details of optimizer implementation. The following chapter contains achieved results of the described scenarios. The last chapter discusses conclusions and the future work.

2. Bobox

Nowadays, increasing performance comes with an increasing number of computational units due the attainment of the physical limits of current technologies. Parallel programming becomes more and more important in the development of performance expensive software to use all the silicon hardware provides. The thread-based approach to achieve parallelism creates a lot of complexity for a programmer to maintain, and it is also not well scalable. It becomes important to abstract the underlying parallel environment to a programmer. The Bobox framework [4, 5] addresses this issue by providing an interface for task-based parallel programming. Such an approach relieves the programmer of handling thread-based programming problems such as synchronization, most technical details (e.g., cache hierarchy, CPU architecture) and communication. Apart from that, task-based programming also allows for better hardware utilization.

According to Bednárek et al. [1], the Bobox framework is *"more useful for data processing scenarios, like database query evaluation or stream processing"*. Without further details, under the hood, the framework has been equipped with a fixed number of worker threads, each one with an own scheduler, and two task queues for every computational unit for a better utilization of CPU caches, using the task stealing mechanism. Communication between tasks uses a column-based data model, the most significant implementation detail that favours data processing problems. Each task has zero or more inputs and zero or more outputs. A single output can be connected to zero or more inputs. A task is scheduled to be executed when it has an unprocessed input.

The Bobox framework provides a C++ library as the interface to its runtime environment. Task granularity is represented by classes derived from the Bobox base class for a task. This base class is called a *box*.

2.1 Design and terminology

The runtime environment handles the implementation details of the task-based parallel environment such as scheduling and the parallel execution of tasks, data transport and control flow. A programmer uses a declarative way to provide the environment with a *model* which defines the way individual tasks are interconnected. The model is used to create a *model instance* which is the base for creating a *user request*. The user request contains only very little additional information compared to the model instance.

After a programmer provides the environment with a user request, he no longer has control over its execution. The framework provides only information about whether it has finished executing the request. The provided user request is divided into individual tasks. When a task is ready to be executed, it is added to the task pool. A worker thread then retrieves the task from the task pool and invokes it.

The basic element of model instance is an element representing a task, called a box. In every model instance, there is a special box called an *initialization box*. This box is responsible for the creation of the initialization input data of all the other boxes in the model. The framework executes this task at the beginning of a request evaluation, and its only goal is to send data to its only output.

Data is sent using an *envelope*, a column-based data structure. An empty envelope is a special type of envelope called a *poisoned pill*. When a box receives a poisoned pill in its input, there will be no more data sent to this input. All the paths of model instances are required to end in another special type of box called a *termination box*. When this box receives a poisoned pill, the execution is finished and the pipeline is deallocated.

2.2 Boxes

Boxes, as the representation of Bobox framework tasks, are executed in three steps.

1. The first step is the *prologue*, when the box creates a snapshot of its inputs and stores this snapshot in member variables so the user code can access it. The prologue communicates with the runtime environment, and synchronization is needed.

2. The second step called *action* is the main place for a user code execution. User code can communicate with the runtime environment using only specific member functions, e.g., it can send an envelope to its output. This approach creates a transparent parallel environment for programmers, relieving them of issues related to the parallel execution.

3. The last step is the *epilogue*. The step handles the scheduling of the next task based on two criteria. A task is scheduled again,

   (a) if it has got an unprocessed input and it has processed some input in the action step.

   (b) if it requested to be scheduled again.

   There is a reason why box is not scheduled again if it has got unprocessed input and it has not processed any input during the execution. It will most likely wait for another input, e.g., the join operation in a database when a task is not executed until it has received data from both inputs. The option to explicitly request another scheduling is there for cases when a single box creates a large output. A task should not run for a long period of time. It can create a large output, and thus congest internal buffers used for communication. A task running for a long time on a single worker thread can also create a bottleneck for a parallel execution when many other tasks wait for the input from this task.

Boxes are main objects of the interest for optimization, because they are the main location for the user code. Based on the static analysis of the action step, additional code can be injected to provide Bobox internal facilities with information about the task.

2.3 Usage

To implement a Bobox task, a programmer has to inherit from the `basic_box` base class. The action step is represented by one of the virtual member functions from Listing 2.1. A programmer is expected to override one of them. The synchronous version is called when all prefetched envelopes are available. The asynchronous version is called when the envelope on the input which is being *listened* is available.

Listing 2.1: The code representations of the box action step.

```
virtual void sync_body();
virtual bool async_body(inarc_index_type inarc);
```

A programmer can also associate names to particular inputs and outputs using helper macros from the Bobox library, see Listing 2.2. The code is easier to comprehend and maintain when box inputs or outputs are referred to by names instead of using indexes.

Listing 2.2: The helper macros for mapping of names to inputs and outputs.

```
#define BOBOX_BOX_INPUTS_LIST(...)
#define BOBOX_BOX_OUTPUTS_LIST(...)
```

The implementation of a task is straightforward for a C++ programmer using only C++ core language features. Unfortunately, C++ syntax is not convenient for expressing a definition of the whole execution model. The language developed to such purpose is called *Bobolang* [6]. Listing 2.3 shows an example of a model definition in the Bobolang language.

Listing 2.3: An example of the Bobolang usage.

```
model main<()><()> {
    bobox::broadcast<()><(),()> broadcast;
    Source<()><(int)> source1(odd=true), source2(odd=false);
    Merge <(int),(int)><(int)> merge;
    Sink <(int)><()> sink;

    input -> broadcast;
    broadcast[0] -> source1;
    broadcast[1] -> source2;
    source1 -> [left]merge;
    source2 -> [right]merge;
    merge -> sink -> output;
}
```

2.4 Cooperative scheduling

Due to the character of the framework scheduler, the user code directly affects the scheduling of tasks and thus the overall performance of an execution. The framework provides ways of manipulating the task scheduling. For example, a task can give up its execution before it finishes naturally. A task can also inform the scheduler that it should not be executed before all the input data is available. Based on the static code analysis of the user code, the optimizer tool can inject function calls into the user code to affect framework scheduling.

3. Static code analysis

Detection of errors in code early in the development process is important for the reduction of development costs. The most commonly used process for detecting errors as soon as possible is called a *code review*.

The static code analysis can be considered to be an automated code review. One of the biggest pitfalls of a code review is its high price. Two or more people read the code looking for a way to improve it, finding and fixing errors, performance issues, or potential errors which can become actual errors in future. The quality of a code review decreases with the time spent reading the code. Developers need to rest in order to increase the quality of their code review.

Compiler warnings can be considered to be a very basic static code analysis. A compiler warns a programmer about suspicious parts of code which it has detected in the compilation process. It is good practice to turn on all compiler warnings and compile the code without any detected. The most commonly used compilers provide a switch to consider warnings as errors. Software development companies often create rules to force programmers to produce a warning-less code, or use the mentioned compiler switch in their development environment to implicitly remove compilation warnings.

Compilers cannot do much to diagnose more complex errors. It is not their primary goal, which is code compilation, and a more advanced diagnostic could increase compilation times. Furthermore, there is no need to produce any binary when analysing code. Most developers of tools for static code analysis would like to access an output of the semantic analysis. It is up to programmers whether they reuse a compiler front-end, or implement their own.

This chapter covers some of possible approaches to creating static code analysis tools. The chapter also describes the implementation of some popular analysis tools.

3.1 C++ tooling

Due to the complexity of the C++ core language, there are only very few fully C++ compliant, open-source and freeware[1] compilers. The two most known are *GCC, the GNU Compiler Collection* [18], and *Clang/LLVM* [14]. Apart from the standard syntax for procedural programming, C++ includes the Turing-complete template meta-programming language. A compiler needs to *execute* code in order to generate code which is eventually compiled into native code.

Based on the given facts, it would be extremely difficult and unwise to individually implement your own C++ front-end. The optimizer tool should use some existing front-end for the static code analysis. The set of available C++ front-ends is very limited, and some of them are not suitable for tool implementation.

[1]Proprietary compilers could do the job just as well, but freeware compilers are preferable.

3.2 GCC - the GNU Compiler Collection

GCC is a compiler with a great 26 years old history and is well established in the C++ software development world. Many helper tools for build environments support GCC in some way, and yet programmers have been struggling with writing static code analysis tools for C++. There are multiple reasons why programmers have not started using GCC. As an example, a citation from *Sparse FAQ* [8] covers their reasons for avoiding GCC:

"Gcc is big, complex, and the gcc maintainers are not interested in other uses of the gcc front-end. In fact, gcc has explicitly resisted splitting up the front and back ends and having some common intermediate language because of religious license issues - you can have multiple front ends and back ends, but they all have to be part of gcc and licensed under the GPL."

The first sentence, especially the first few words, is the main reason programmers have not started using the GCC front-end to create tools for a static code analysis. Some of the disadvantages in using GCC for tooling are:

- It is very hard to learn for beginners.
- Even though GCC consists of front-end, middle-end and back-end, it still appears to be monolithic. It is very difficult to decouple front and back ends.
- GENERIC and GIMPLE[2] representations of code are not intuitive.
- GCC does not keep track of tokens locations in source code, e.g., it does not keep track of macro expansions. Therefore, it is very difficult to refactor the code correctly.
- The code is optimized when it is parsed so the abstract syntax tree does not correspond to the source code, e.g., `x-x` is optimized to be `0`. It is extremely difficult to refactor the code based on such an optimized abstract syntax tree.

3.3 Elsa: The Elkhound-based C/C++ Parser

Even a smaller group of developers is able to create a relatively nice C++ compiler front-end. *Elsa* [20] is such an example. It provides a programmer with a user-friendly AST representation of code, which is designed to be easily extensible without writing a single line of C++ code. The front-end provides a mechanism designed as the *visitor pattern* for the AST traversal. The other way to traverse a tree is to traverse the tree manually. The visitor pattern is useful for the context-insensitive traversal.

The biggest disadvantage of Elsa is that its development stopped long time ago, in 2005, when a different project called *Oink* [21] started. Oink uses the Elsa

[2]GENERIC and GIMPLE are names for different representations of AST. GIMPLE is a subset of GENERIC for code optimizations.

front-end. Later, the Oink development stopped before the year 2011 when C++ experienced its *renaissance* with the new approved standard, which introduced big changes to the core language and the library. Therefore, there is almost no support for new C++11 language features. Oink, just like Elsa, does not have an integrated preprocessor so it is extremely difficult to map AST with locations in source code. Elsa also suffers from a lower speed, but this would only be a negligible disadvantage for smaller projects.

3.4 VivaCore/OpenC++

The *VivaCore* [23] library was developed as the basis for the *PVS-Studio* [22] static code analyser for C/C++. The library is derived from the older *OpenC++ (OpenCxx)* [25] library. The idea of using OpenC++ appeared when the team was implementing the *Viva64* [23] library. They were making many changes to OpenC++ and because of the lack of resources, they did not continue to improve the library[3]. Instead, they developed their own library. The VivaCore library has become popular. It has been used as a base by other popular tools such as *VisualAssist* by *Whole Tomato Software*, *Doxygen*, *Gimpel Software PC-Lint*, *Parasoft C++test* and others.

Figure 3.1 shows the library design. The library uses an external preprocessor. Without an integrated preprocessor, it is extremely hard to track macro expansions and the actual locations of symbols in source code. Thus, source-to-source transformations are correspondingly difficult.

A preprocessed input is passed to the library. Two library subsystems process the code before it reaches the lexical analysis. The first is the input subsystem responsible for putting preprocessed code into internal data structures. Internally, the second subsystem is called the *Preprocessor*, but it does not preprocess an input in the meaning of the C++ preprocessor. It is responsible for two operations:

- Splitting the code into strings and separating them into two logical groups. One group is for system libraries, the other group is for user code. Library users can choose whether they want to analyse the system code or just the user code
- Removing compiler specific constructions not related to C/C++ languages, e.g., `SA_Success` and `SA_FormatString` are present in Visual Studio‘s headers.

The next step is the lexical analysis. An output of *Lexer* can be used for basic metrics or syntax highlighting. VivaCore allows modifications to the set of tokens for the lexical analysis.

VivaCore provides a user with *parse tree (PT)*, called also *derivation tree (DT)*, as an output of the syntactic analysis. The parse tree differs from an abstract syntax tree in the way it contains nodes representing the derivation rules used in the syntactic analysis. The word *abstract* comes from the reasoning

[3]Many changes did not fit into *"general OpenC++ ideology"* [24] so they would need to adapt and allocate new resources for such process.

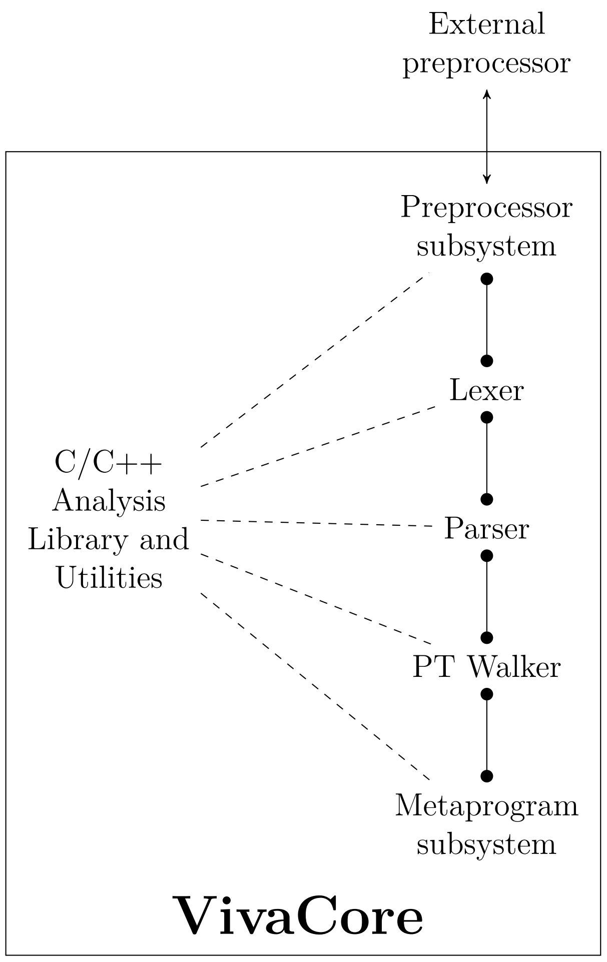


Figure 3.1: The design of the VivaCore library.

that the structure hides the rules used in its construction. It is actually possible to traverse PT as if it was AST. VivaCore‘s PT defines two basic sets of nodes with ancestors in `NonLeaf` and `Leaf` base classes, which have `PTree`[4] class as their common ancestor declaring the only pure virtual member function. It is the only function which must be overridden in inherited classes, allowing their design to be more flexible.

Probably the most interesting part of the library interface is tree traversal. Three different *walker* classes have been implemented for this purpose.

Walker is responsible for walking over basic C++ constructions.

ClassWalker handles C++ class specific features.

ClassBodyWalker traverses a body of a C++ class.

It is possible to traverse PT multiple times. Users can traverse the code for measurements at first. Later, in further traversals, they may modify PT. The modification of tree nodes can trigger a tree rebuild.

[4]`PTree` has `LightObject` as its base class used in GC.

The VivaCore library is one of the best libraries for the static analysis of C++ code. A relatively high number of very popular tools for code analysis based on the library reflects its quality. Its development is still in progress and the library is still being updated. Developers have also implemented support for new features introduced to new approved language standards. However, the goal of the optimizer is not only to analyse code, but also to transform user code. An external preprocessor is a major issue in using the VivaCore library.

3.5 Static code analysis tools

Code quality in large projects is hard to maintain using only a code review. If there are many code related commits every day (e.g., *Crysis 2* multiplayer had ~100-150 code related commits every day collecting 130 different developers over the last year of the development [9]), providing human resources for a code review would be inefficient. Instead of that, companies use tools for a static code analysis and the diagnostic is reviewed further. However, not many companies trust tools enough to let them perform source-to-source transformations, apart from formatting or simple refactoring.

This section describes some popular static code analysis tools, in order to familiarize a reader with solutions already used to implement such tools. Even though their final goal differs from the goal of this thesis, they all have to achieve the common goal of *understanding* source code to some extent.

3.5.1 Clang Static Analyzer

The static analyser is a part of the Clang project implemented on top of the Clang tooling interface. The analyser is easily extensible by implementing *checkers*, even though their interface may not be intuitive. Authors demonstrate how to write a simple checker for Unix stream API in the presentation called *"How To Write a Checker in 24 Hours"* [10]. When writing a checker, the developer needs to understand how the analyser works under the hood.

The core of the analyser performs a symbolic execution of the code, exploring every possible path, tracking all the variables and constructing a *Control Flow Graph (CFG)*. Checkers participate in CFG construction. Essentially, checkers are visitors which react to a specific set of events while traversing AST (e.g., `checkPreStmt`, `checkPostCall` functions) and eventually creating new CFG nodes.

The analyser aims to solve path-sensitive problems, e.g., problems related to the resource acquisition and release, such as resource leaks and resource usage after release. The CFG construction is the core of such analysis. Actually, the development manual page of the analyser contains important advice which discourages developers from implementing path-insensitive checkers [11]:

"Some checks might not require path-sensitivity to be effective. Simple AST walk might be sufficient. If that is the case, consider implementing a Clang compiler warning. On the other hand, a check might not be acceptable as a compiler warning; for example, because of a relatively high false positive rate."

3.5.2 Clang Format

Consistency in code formatting is very important in large projects. It increases readability and the code becomes better machine-editable. Even though consistent code formatting is very important, there are not many tools which support automatic code formatting for C++, e.g., *BCPP*, *Artistic Style*, *Uncrustify*, *GreatCode*, *Style Revisor*.

The reason why companies allow the usage of automatic formatting tools is that those tools guarantee they will not change the code semantic, i.e., they edit only white space characters, literals and comments. Therefore, they will not break a compilation. There was a proposal to let clang-format reorder file includes, but it was not approved because such a change can break a compilation. The main challenges for clang-format developers based on their design document [12] were:

- A vast number of different coding styles has evolved over time.
- Macros need to be handled properly.
- It should be possible to format code that is not yet syntactically correct.

It was a hard decision for clang-format developers whether they use lexer or parser to implement such a tool. Both have their advantages and disadvantages in terms of performance, macro management or type information. In the end, they decided to keep the implementation based on lexer, but there is still an ongoing discussion about adding AST information. However, this discussion is leaning towards creating a separate tool using AST, which already has the name *clang-tidy* [26].

3.5.3 OCLint

OCLint [27] is a tool implemented on top of the Clang tooling interface. It tries to create a generic framework for code diagnostic. Main parts of OCLint are *Core*, *Rules* and *Reporter*.

Core controls a flow of the analysis, dispatches tasks to another modules and outputs results. It parses code, builds AST and provides modules with access to this AST. While parsing code, it creates various metrics such as:

- Cyclomatic complexity.
- NPath complexity.
- Non-commenting source statements.
- Statement depth.

Rules can provide *RuleConfiguration*, which defines limits for metrics. When limits are exceeded, *Core* emits a violation. There are two main approaches for the modules in handling the diagnostic:

Line based is when modules are provided with lines of code.

AST based provides modules with an access to AST using two approaches:

- Using the *visitor design pattern* to explore AST.
- Defining *matchers* for suspicious code patterns.

Modules are separated from *Core* code, so they can be loaded in runtime. The basic diagnostic can be represented as a set of code patterns. The last task which must be performed is reporting a discovered diagnostic using *Reporter*.

However, pattern matching is not a mechanism strong enough to catch even a slightly more complex error such as a resource leak. Supported approaches other than AST matchers do not really help more than just using the Clang tooling interface directly.

3.5.4 Cppcheck

An example of a tool which does not use any compiler front-end to process source code is Cppcheck [28]. The tool performs the code parsing and the analysis on its own, but the quality of understanding the source code is lower than in well-established compiler front-ends. The input for code checks is the output of the lexical analysis. Thus, it can be very difficult to implement more advanced checks. The fact that the code analysis passes only the lexical analysis phase also means that the tool is not even able to catch syntactic errors. Fortunately, Cppcheck implements classes such as `Scope` or `SymbolDatabase` with the functionality which their names indicate.

The simplified version of a Cppcheck execution from documentation for programmers can be written in eight steps [13]:

1. Parse the command line.
2. `ThreadExecutor` creates necessary `CppCheck` instances.
3. `CppCheck::check` processes every file.
4. Preprocess a file inside the `check` member function.
   - Comments are removed.
   - Macros are expanded.
5. Tokenize a file using `Tokenizer`.
6. Run all checks on the `Tokenizer` output called a token list.
7. Simplify a token list.[5]
8. Run all checks on a simplified token list.

[5]There are various simplifications applied to a token list. Every simplification passes the whole token list looking for patterns and potentially changes this list. For example, the first applied simplification changes `"string"[0]` to `'s'`. Another example is removing of `std::` tokens from a specific set of function calls.

3.5.5 Summary

Three out of four described code analysis tools are implemented on top of the Clang front-end. Cppcheck parses code by itself and cannot be considered to be a full-featured front-end. Cppcheck understands code enough to implement simple checks, but more complex analysis could become very difficult to implement.

Nonetheless, each one of the three analysis tools implemented on top of Clang has a different goal and a different *view* of code from the other two tools. Such a variability indicates that the Clang front-end exposes a lot of information gathered during a compilation to tools. Thus, its tooling library is a good candidate for the library used in implementing the optimizer.

3.6 Related work

There are not many tools for front-end optimization. The main reason is that it has been difficult to implement any front-end tool in general. Furthermore, most optimizations have already been implemented in compilers. However, this thesis aims to optimize the specific framework.

3.6.1 Scout

The front-end optimizer tool, called *Scout* [29], is being developed in *TU Dresden*. It is supposed to do transformations for front-end SIMD optimizations, e.g., loop auto-vectorization, a very similar task to what most current compiler back-end optimizers do. It will transform C code into optimized C code with compiler intrinsics. Naturally, auto-vectorization is done by a compiler back-end optimizer, but there are limits to what the compiler can do. It needs to use the extensive dependency and alias analysis to verify the correctness of the vectorization and often rejects more complex loops. Some compilers allow programmers to annotate loops with `pragma` directives, leaving programmers responsibility for keeping some loop invariants. A compiler can skip those checks before vectorization, thus accepting more loops. Unfortunately, the measurement with the specific Intel compiler using pragma directives gave insufficient results. For example, the compiler rejected loop vectorization after the loop variable type was changed from `unsigned int` to `signed int`. Actually, Scout provides a semi-automatic vectorization, where programmers have to annotate loops using pragma directives to enable the vectorization of a given loop.

The tool provides a command line interface as well as a graphical user interface. It uses Clang to build AST from C code. AST is then transformed into a different AST which represents optimized code. Finally, this optimized AST is transformed back to C code. The tool can be configured with a set of used intrinsics, i.e., *SSE2*, *SSE4*, *AVX*, *AVX2* or *ARM NEON*.

3.7 Summary

All the tools for a static code analysis described in this chapter were implemented by a group of programmers. Without a good library for parsing C++ code, implementing any tool for an analysis of C++ code is an extremely difficult task.

The requirement for a code transformation makes the task even harder. The lack of the C++ tooling support has been generally mentioned by programmers as being one of the biggest drawbacks of using the C++ language. The situation has changed with the increased support of tooling at the Clang front-end and it is possible to implement a relatively complex tool for front-end optimizations individually or using a small group of people. The Scout tool is an example of this.

The Clang front-end has intentionally been omitted from this chapter, as the optimizer tool uses its tooling interface to analyse and transform code. The whole of the next chapter covers the Clang tooling interface in detail. The Scout tool is the main inspiration for the decision to use the Clang front-end for analysis and transformation since the tool has a very similar goal and its major part is implemented by a single programmer.

4. Clang and tooling

A support for creating static code analysis tools for C++ was very subtle before Clang developers increased its support for tooling. All compiler front-ends were cumbersome to use, and implementing a new front-end individually is extremely difficult. The situation has changed with Clang providing API for access to C++ code represented as the user-friendly abstract syntax tree structure. Actually, Clang does not provide single tooling API, instead it provides multiple APIs with differences in usage. The differences mainly affect the way in which a tool accesses AST, the range of accessed information, and compatibility with older versions. Tool developers can decide whether they want to sacrifice compatibility to all the information provided by front-end internals. As Clang is being developed, there is no guarantee that interfaces in their code base will not change.

The tooling interface indicates that Clang focuses on diagnostic, code completion and refactoring tools. The support for source-to-source transformation is subtle. Even though there are multiple ways to transform code, most of them are deprecated.

4.1 Abstract Syntax Tree

The structure Clang provides is not only an abstract syntax tree of code. It is a graph with AST nodes, but with more edges than those in AST. Clang provides mechanisms for traversing this graph as if it was AST. With access to the AST node, programmers are able to traverse a graph in more ways than they would be able to do with just the AST structure. It allows developers to optimize their analysis code or perform a more complex context-sensitive traversal.

Unusually, the class hierarchy of nodes does not have a common ancestor. There are two large hierarchies with common ancestors in `Decl` and `Stmt` classes, some important ones with ancestors in `Type` and `DeclContext` classes, and many classes accessible only from specific nodes.

4.1.1 Traversal

The template responsible for the AST traversal is called `RecursiveASTVisitor`. It is implemented as a *Curiously Recurring Template Pattern (CRTP)* combined with the *visitor design pattern* where a programmer is able to either react on the AST node visit or manipulate a traversal. Due to the character of the AST nodes class hierarchy, the implementation of the visitor template is cumbersome, with the extensive usage of macros. Therefore, the template has been nicknamed *macro monster*. It has been promised that it will be reimplemented some time so there is no guarantee that the interface will not change, even though the visitor is widely used in tools. The other approach for traversing AST is to follow the edges. It is more useful in a context-sensitive traversal.

4.2 Source-to-source transformation

Even the simplest case of a code transformation such as symbol renaming is difficult to implement in C++. Before the lexical analysis starts, with some exceptions[1], there is the text-preprocessing phase when source code text is transformed into a different source code text. A preprocessor does not know anything about language syntax or semantics, it is defined as a set of operations on text. During the preprocessing phase, symbols may be created, copied, or erased. Tracking the symbol's origin from the output of the syntactic analysis to a source code location before preprocessing is a difficult task. Clang uses an integrated preprocessor, so looking up a source location from AST is simpler than it would be with an external preprocessor.

There are multiple approaches to perform source-to-source transformations. If a tool is supposed to support operations such as symbol renaming or code completion, Clang allows a programmer to rewrite source code as text. The `Rewriter` class provides such functionality. For greater control over code changes in specialized tool wrappers, there is the `Replacement` class. Furthermore, if a tool is supposed to be used in a build process, the best solution is to transform AST and output code from this transformed AST for the next build step, see Section 4.2.3.

4.2.1 Rewriter

For basic source code transformations on the level of text editing, there is the `Rewriter` class. A programmer can create as many instances as necessary, passing them just a reference to `SourceManager`. A user is then allowed to do operations such as an insertion, a removal or a replacement of text using `SourceLocation` or `SourceRange` objects. Both objects can be gathered directly from most of the AST nodes. Text transformations are far from ideal in C++. However, it is sufficient for renaming symbols or code completion in text editors, where a programmer can immediately repair any compilation errors caused.

4.2.2 Replacements

The special wrapper for greater control over operations in the `Rewriter` class is called `Replacement`. The callback function in the AST matchers interface is provided with `Replacements`, which is a container of `Replacement` objects. The callback is free to manipulate this set, e.g., mainly by adding new objects, but it is not prohibited from removing or editing existing items. At the end of an analysis, a tool tests whether the analysis has finished correctly. Then it checks `Replacement` objects for validity and if all tests pass, the tool applies those objects to the `Rewriter` object. After all the `Replacement` objects have been successfully applied to the `Rewriter` object, the last step is to save the affected files.

A programmer should implement all mentioned steps for the correct usage of the `Rewriter` class. The problem arises when a developer wants to refactor a code with compilation errors. `RefactoringTool` will not save any changes when

[1]The token paste operator `##` must be handled when the lexical analysis is happening.

a compilation fails. Thus, refactoring tools integrated to a source code text editor cannot use these facilities.

4.2.3 TreeTransform

The most correct approach to AST transformations, according to Clang developers, is to use the `TreeTransform` class. If a programmer has access to mutable nodes, they often provide member functions for a manipulation with *edges* to other nodes. The Scout tool (Section 3.6.1) manipulates AST nodes and edges directly, even though Clang developers deprecate the direct manipulation of AST. The problem is that nodes and edges actually create a more complex structure than AST. It is difficult to manipulate this structure without having a detailed knowledge of it, i.e., a knowledge at the level of a Clang developer. A developer has to know the lifetime of a node and all the possible edges coming to and from a node. It is not advisable to try to modify AST manually.

On the other hand, Clang itself internally transforms AST multiple times during a compilation process. For example, a template instantiation is done on the constructed AST, effectively transforming it into a different one. Since a template instantiation can break the code semantic, the newly-created AST must be tested in the semantic analysis represented by the `Sema` class. This process is handled by the `TreeTransform` class. Even though its interface is simple, using the CRTP pattern, it is hard to use `TreeTransform` in tools. None of Clang tooling interfaces provide access to the `Sema` object which is necessary for the construction of the `TreeTransform` object.

4.3 LibClang

The first mentioned, but the least suitable tooling API for achieving the thesis goal is LibClang [30], a library with an interface in the C language. Its major advantage presented by Clang developers is that it is supposed to be relatively stable and backward compatible. For some developers those features can be crucial, but they are not important for achieving the thesis goal.

Even though LibClang provides an interface in a different language than Clang internals, it does not try to hide the way code is represented there. It provides access to Clang AST[2] in the form of an abstraction called *Cursor*, which represents a single AST element. A tree traversal is achieved using the *visitor design pattern*. A part of the library supports code completion, so the library fits well as a basis for source code text editors tools.

4.4 Plugins

Clang allows a developer to step into a compilation process in the form of plugins [31], dynamic libraries loaded in runtime and running their actions on processed code. It is simple to integrate the plugins into a build environment where Clang is used as the compiler. They can be used to break a compilation (e.g., coding rules are broken) or they can produce some output (e.g., code statistics).

[2]It is necessary to mention that access is very limited relative to Clang internal AST.

Because plugins are already a part of a single compilation step, they are not suitable for source-to-source transformations. Unlike when using LibClang, the developer of Clang plugins has full access to AST.

Even though the purpose of this thesis is to create a tool used mainly in a build environment, it should not be limited to environments where Clang is used as the compiler, or to force build environments to integrate Clang in any way.

4.5 LibTooling and AST matchers

LibTooling [32] aims to write standalone tools such as checkers or refactoring tools. It is easier to run a standalone tool on a single file or a specific set of files. On the other hand, it is harder, but definitely possible, to integrate such a tool into a build environment where it can be triggered by dependency changes.

The library interface provides a developer with full access to the AST structure. Even though the interface tries to hide other compiler internals, it is a part of the compiler code and the developer has access to them. The developer can take advantage of other powerful facilities in the compiler such as `Lexer`, `Parser`, `Sema`, `SourceManager` or `TreeTransform`.

AST matchers [33] aim to solve the very fundamental operation of matching patterns in AST. Most tools do not invoke an action on every single node in AST, rather they invoke an action only on specific nodes, e.g., nodes representing a member call expression on a specific class. Without AST matchers, a programmer has to traverse a whole tree looking for patterns, and eventually invoke an action on matching nodes. Clang provides an extensive library of matcher classes which are designed to be combinable. For example, matchers for the `if` statement and the function call expression can be combined into the matcher for the `if` statement where the condition is a function call expression.

Both libraries are part of Clang source code and unlike LibClang, these libraries do not abstract internal compiler structures. They only represent the way those structures are accessed. Therefore, both libraries can be used interchangeably, e.g., developers can use AST matchers to seek nodes in AST, and then they can run a front-end action on a sub-tree using LibTooling.

4.5.1 Internals

Every compiler front-end uses some powerful facilities in the compilation process. With access to these facilities, a developer has access to more information about source code. More information allows the implementation of more complex algorithms. If compiler internals are accessed before their invocation, the tool can also affect the compilation process.

Preprocessor

The `Preprocessor` module closely cooperates with the lexer in the transformation of source code text into lexical tokens. The `Lexer` class should see the code as a single source file. It should not handle code preprocessing actions such as resolving file includes and macro expansions. The integrated preprocessor makes Clang tooling libraries more suitable for the implementation of tools which perform source-to-source transformations than other libraries. The integrated preprocessor allows better tracking of macro expansions and searching for source code locations from AST nodes. Some useful information provided by the Clang preprocessor is:

- A list of all predefined macros.
- Access to an immediate macro name for a source code location.

Lexer

The `Lexer` class provides a simple interface for the transformation of the text buffer into the stream of tokens. Only forward lexing is supported. The class provides:

- The source location just past the end of the token specified by the provided source code location.
- The token string for the provided source location.

Parser

The compiler parser is implemented in the `Parser` class. The class implements the parser for the C family of languages, i.e., C, Objective C, C++ and Objective C++. Clang implements its own hand-written recursive-descent parser as several other C and C++ front-ends do[3]. The recursive-descent implementation and the complexity of the C++ grammar makes `Parser` a relatively large class in terms of member functions count. However, it is not an interesting class for tools. The majority of member functions handle the grammar rules resolution.

Sema

`Parser` feeds the `Sema` object with information using the `Action` interface. Essentially, `Parser` notifies `Sema` when code is being parsed. Based on notifications, the `Sema` object constructs the AST structure. After an entire translation unit is parsed, the `ActOnEndOfTranslationUnit` action is invoked and `Sema` provides `ASTConsumer` with constructed AST. This is the point where plugins and LibTooling libraries start a code analysis by providing own `ASTConsumer` implementation through the `FrontendAction` interface.

`Sema` is one of the most interesting classes for tools from outside of the AST library. It provides information related to:

- Name lookup.
- Semantic checks.
- Code completion.

[3]GCC used the generated Bison/YACC parser, but authors implemented own hand-written parser in the end. Elsa uses the recursive-descent parser as well.

SourceManager

The class essential for tools performing source-to-source transformations is called `SourceManager`. It is responsible for source code management on top of a filesystem. It handles the loading and caching of source code. Furthermore, the class is able to translate abstract `SourceLocation` objects into *spelling* and *expansion* locations. A spelling location is a location where bytes for a specified token come from, and an expansion location is a location where a programmer can see them. For a macro expansion, a spelling location is a location in a macro definition, and an expansion location is a location where a macro was expanded. `SourceManager` provides some useful information such as:

- Spelling and expansion line and column numbers.
- Whether a location is in a system header.
- Whether a location is in the main translation unit file.
- Whether a location is in a macro expansion.
- The memory buffer for translation unit source code.
- Various macro expansion information.

4.5.2 Usage

Clang, just like most other compilers, can receive compilation options such as predefined macros, include directories, forced includes or a diagnostic level as command line arguments. A standalone tool must be able to feed the compiler internally with this kind of data. Since a tool can have its own command line arguments, it would be hard to distinguish tool and compiler arguments. LibTooling tools gather compilation options from a file with the special name `compile_commands.json`. A tool tries to lookup the file with this name in the parent directories of a currently compiled file. If it succeeds, it uses the file to build a compilation database.

4.6 Optimizer implementation

The LibClang‘s advantage in the backward compatibility and stability is negligible in achieving the optimizer goal. Even though LibClang creates a new layer on top of compiler internals, it still provides enough information to implement the optimizer tool. However, access to information provided by the compiler internals could allow the optimizer to implement more complex algorithms or new optimization methods.

Clang plugins limit the tool to environments with the Clang compiler. Clang plugins also cannot be interactive. They cannot interrupt the compilation process to wait for a user input. An easy integration into a build environment is an obvious advantage.

LibTooling and AST matchers libraries provide the possible implementation of the standalone tool. Furthermore, such tool has access to compiler internals.

The drawback of the backward compatibility is not an issue, since any interface changes can be handled easily, as the tool is not expected to have a large code base. The stability depends on the stability of Clang libraries. One only needs to follow the Clang development more closely. The real issue lies in the integration of the new tool into a build environment.

In overall, LibTooling and AST matchers libraries provide more advantages than disadvantages for the development of the optimizer tool. Furthermore, Clang plugins, LibTooling and AST matchers libraries use the same code base. Therefore, building the tool as a Clang plugin requires a small amount of specific code that differs from the code used to build the optimizer as a standalone tool.

5. Prefetch method

Tasks in the Bobox framework are represented in form of boxes, which can have zero or more inputs. The boxes are elements of a model. Based on terms defined in Section 2.1, before an execution of the model, a model instance is created, later decomposed to tasks, which are then scheduled and executed. The problem is that the scheduler lacks information about a box execution, specifically about processing of its inputs. There are three cases of input data requirements for a meaningful task execution:

1. A task does not need data from any input at all.
2. A task needs data only from some inputs out of multiple inputs.
3. A task needs data from all inputs.

An execution of a task from the second and the third case without necessary input data adds significant overhead to a model instance execution. Scheduling itself does not have a negligible overhead. Synchronization is necessary before and after the task execution. If a task is executed before it has all necessary input data, it finishes its execution immediately. In such case, scheduling consumes all CPU time.

However, a developer can provide the scheduler with information about the necessity of data from a specific input using the `basic_box` base class member function. All overloads of this member function are listed in Listing 5.1.

Listing 5.1: `basic_box` prefetch member function overloads.

```
bool prefetch_envelope(input_index_type input,
                       unsigned count = 1);
bool prefetch_envelope(input_index_type input,
                       inarc_index_type offset,
                       unsigned count = 1);
bool prefetch_envelope(inarc_index_type inarc,
                       unsigned count = 1);
```

The function informs the scheduler about a number of envelopes on a specific input necessary for a meaningful box execution. Ideally, a programmer with a good knowledge of the box design adds function calls with correct values to the code.

For the case where a single envelope from all inputs is necessary, the good design solution is to use class inheritance and implement a common base class that calls the prefetch member function for all its inputs. Programmers need to remember that they implement this special case of a box and they should derive from this base class. Class inheritance may not be as useful for cases where only data from some inputs is necessary. On the contrary, using class inheritance to achieve code reuse in these cases is a bad design.

The goal of the optimizer is to search for a usage of box inputs and inject prefetch member function calls accordingly.

5.1 Restrictions to optimization

The optimization cannot be applied to the source code when some restricting conditions are satisfied. The algorithm for the prefetch optimization does not produce any runtime checks, but the static analysis checks various conditions whether it is safe to apply changes to the source code. Firstly, the analyser tests a box class for various conditions whether it can be optimized at all, then it tests all box inputs one by one for another set of conditions. If a box and its input pass all tests, the box input is prefetched. Therefore, some restrictions can completely inhibit the box optimization, some of them can inhibit the optimization of a single input.

The optimization of a box is discarded if at least one of these restrictions is satisfied:

(global.1) There are no functions with the user code for the action step, see Section 2.2, i.e., a class does not override any of functions representing the action step listed in Listing 2.1.
Rationale: If there is no user code in a class, there is no usage of any input in a context of this class. Improbable case, but it has to be taken into an account.

(global.2) There are no inputs.
Rationale: Nothing to optimize.

(global.3) There is no mapping of names to inputs created by using the Bobox helper macro, see Listing 2.2.
Rationale: Currently, the optimizer identifies inputs by names associated to them by using the Bobox helper macro. If there is no such mapping, the optimizer does not detect any input on a box[1].

(global.4) A definition of the overridden `init_impl` member function is inaccessible.
Rationale: This member function represents the initialization step of a box execution and it is the location for prefetch calls. If the analyser cannot access its definition, there is no place to put function calls. The definition may be inaccessible due to various reasons such as it is defined in a different translation unit.

(global.5) The corresponding `init_impl` in the base class is private.
Rationale: The analyser is able to override the initialization member function, but a programmer may assume that the corresponding initialization function from the base class is called. Therefore, there has to be a call to the base class corresponding function in the newly overridden function definition. However, if the function is inaccessible due to the protection level, the function call would break a compilation.

[1]Since the optimizer does not implement any complex constant expression evaluation, it is expected that programmers use a named helper to refer to an input or output rather than a numeric constant.

Single input optimization restrictions:

(single.1) There is already the prefetch call for the input.
Rationale: A programmer already handles the optimization.

(single.2) The optimizer cannot detect whether data from the input is likely to be necessary.
Rationale: The decision to prefetch such input is as good as the decision not to. It can happen when data from an input is necessary only in a single branch of the code or not at all.
Note: The important word in the restriction wording is *likely.* The analysis does not have to *prove* that data from the input is necessary rather just *assume.* The requirement for the proof that data from the input is necessary would inhibit a big portion of possible optimizations. For example, such an assumption can be that a loop body is executed at least once.

5.1.1 Overriding initialization step

Prefetch calls are placed in the box initialization step. If there is an accessible implementation of the initialization function, prefetch calls are injected into this definition. If the initialization function is not overridden, the optimizer is able to inject the overridden implementation by itself. The problem with an injection of the completely new overridden initialization function is that the previously overridden initialization function can prefetch inputs itself. Fortunately, if the prefetch call on the same input is called multiple times, only the last call has effect as it overrides the previous call. Therefore, the injected function calls prefetch functions on the beginning of the definition and the call to the previous overridden initialization function as the last statement, see Listing 5.2.

Listing 5.2: The generated box initialization function definition.

```
virtual void init_impl()
{
    // prefetch_envelope for desired inputs
    some_base::init_impl();
}
```

Calling the previous corresponding `init_impl` function as the last statement ensures that if there is a prefetch call, it is the one that counts.

5.2 Searching for values in code

To check the restriction *(single.1)*, the analyser must search for prefetch calls on inputs in a code likely to be executed in the box initialization step. Furthermore, the restriction *(single.2)* describes searching for a usage of a box input in the box action step. Basically, the analyser must search for values[2] that are present on all paths or paths likely to be executed in *Control Flow Graph (CFG)* of a specific

[2]A value is a too abstract notion. For example, such a value can be the name of a callee in a call expression represented by the `CallExpr` AST node.

function definition. Clang tooling libraries provide a developer with AST, but it is also possible to construct CFG using Clang static analyzer code, see Section 3.5.1.

Fortunately, the construction of CFG from AST is not necessary since a slightly modified default AST traversal can achieve the same result. Section 4.1.1 related to the AST traversal mentions that a developer can *override* a tree traversal when using the visitor pattern approach. In more details, the `RecursiveASTVisitor` template provides member functions with names starting with `Traverse*`[3], which are responsible for a traversal of the internal graph structure. Actually, these functions are responsible for traversing the structure kept internally in Clang as if it was AST. Those member functions can be *overridden* using CRTP.

Figure 5.1 and Figure 5.2 show an example of a traversal of the same code in CFG and AST structures. Figure 5.1 shows CFG of the code with a single `if` statement with non-empty then and else branches followed by a non-empty block. B represents the condition expression block, B1 and B2 represent then and else branches of the `if` statement, and C is the last non-empty block on both paths from *Entry* to *Exit* blocks. Figure 5.2 shows the AST representation of the same code combined with nodes and edges from Figure 5.1 with the simplification that `IfStmt` is followed by the block C in the `CompoundStmt` node. *Entry* and *Exit* nodes and dashed edges do not exist in AST. The only shared edges between CFG and AST are dashed-dotted edges from B to B1 and from B to B2. For example, if the analyser searches for a value on the path passing through block B1, assuming it starts in `CompoundStmt`, it visits node by node in the graph depth-first search algorithm:

1. `CompoundStmt`
2. `IfStmt`
3. *Block B*
4. Returns to `IfStmt`
5. *Block B1*
6. Returns to `IfStmt`
7. Returns to `CompoundStmt`
8. *Block C*
9. Returns to `CompoundStmt` and finishes

The exactly same sequence of code blocks that would be searched in CFG: block B, block B1 and block C. *Entry* and *Exit* blocks are empty thus not interesting for the optimization process.

[3]* represents a type of an AST node such as `TraverseStmt` for a statement or `TraverseCallExpr` for a call expression.

$$\begin{tikzcd}
 & & B1 \arrow[dr] & & \\
\textit{Entry} \arrow[r] & B \arrow[ur] \arrow[dr] & & C \arrow[r] & \textit{Exit} \\
 & & B2 \arrow[ur] & &
\end{tikzcd}$$

Figure 5.1: The CFG representation of a code with a single `if` statement.

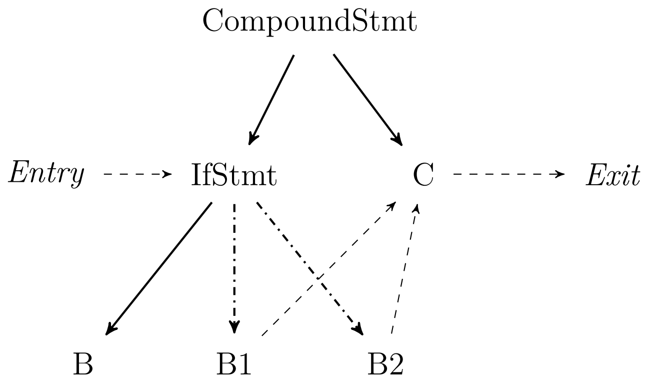


Figure 5.2: An example of a tree traversal.

5.2.1 Divide and conquer

When searching a value in CFG, it would be necessary to either traverse the same path multiple times or remember which nodes and paths were already processed. On the other hand, divide and conquer algorithm design paradigm fits perfectly to the described custom AST traversal.

The implementation of the search algorithm in the optimizer tool enhances `RecursiveASTVisitor` functionality as it has already the well-established interface using the widely known pattern. The problematic part is to identify which AST nodes can affect a control flow of a program and handle their traversal in the implemented template. There are relatively many classes for AST nodes. However, sections **5 Expressions** and **6 Statements** in the C++ standard [7] cover all constructs that can affect a control flow. Statements and expressions that affect a control flow are listed in Figure 5.3. Both sections from the C++ standard can be relatively precisely mapped to Clang AST nodes in the `Stmt` class hierarchy and its `Expr` sub-hierarchy.

Searching for a value in a linear program flow is straightforward. The algorithm visits node by node testing whether it contains a searched value. If the search algorithm encounters a selection statement, it runs itself on every branch. If a searched value is found in all branches, it is found for a current selection statement. If it encounters an iteration statement, it can continue searching in a loop body based on a tool configuration. Jump statements stop searching. A value is searched only in the left-hand side expression of logical expressions because of a short-circuit evaluation.

- **5** Expressions
  - **5.14** Logical AND operator
  - **5.15** Logical OR operator
  - **5.16** Conditional operator
- **6.4** Selection statements
  - **6.4.1** The if statement
  - **6.4.2** The switch statement
- **6.5** Iteration statements
  - **6.5.1** The while statement
  - **6.5.2** The do statement
  - **6.5.3** The for statement
  - **6.5.4** The range-based for statement
- **6.6** Jump statements
  - **6.6.1** The break statement
  - **6.6.2** The continue statement
  - **6.6.3** The return statement
  - **6.6.4** The goto statement
- *try-block*

Figure 5.3: Expressions and statements that affect a control flow.

5.2.2 Loop with fixed number of iterations

It was already mentioned that loop bodies are searched for values by default since they will likely be executed. But this option is configurable in the optimizer tool. A user can choose to disable search in loop bodies that cannot be proven to be executed at least once.

The simple case of a loop where it can be proven that its body is executed at least once is a `for` loop with a fixed number of iterations which was widely used in an old C code, see Listing 5.3.

Listing 5.3: A `for` loop with a constant number of iterations.

```
for (int i = INIT_CONSTANT; i < COUNT_CONSTANT; ++i) {...}
```

If the analyser can prove that `i` is not modified in the initialization statement and the condition expression, it can evaluate the condition as a constant expression. The tool is implemented on top of the compiler, which already has facilities necessary for operations such as the constant expression evaluation or the constant unfolding optimization. Clang exposes functions related to the constant expression evaluation in the `Expr` class. For example, it can evaluate an expression as a boolean condition, but it succeeds only if an expression is really constant for the compiler, which is not in this case. The tool can trick the compiler by setting temporarily the variable initialization declaration to be a constant expression. The same trick can be used to analyse even more complex loops, see Listing 5.4.

5.2.3 Exceptions

The *try-block* statement in the list in Figure 5.3 deserves a more detailed description. Exceptions are a powerful language mechanism which can change a control flow at almost any time. The algorithm for searching values in a code recognizes

```
bool loop = true;
/* loop is not modified */
while(loop)
{
    ...
    if (condition) loop = false;
    ...
}
```

Listing 5.4: Another example of a loop with at least one body execution.

them to very little extent. It searches in a try-block statement and ignores catch statements. Catch statements represent handles of a program in an erroneous state, a state which is not expected to happen, and its transition to the normal state.

5.3 Searched values

The previous section describes how values, what is a bit abstract notion, are searched on all paths in CFG. This section describes what values are searched and reveals other abstract notions from Section 5.1 such as *"input is likely to be used"*.

5.3.1 Available inputs

Inputs in box member functions are referred using `input_index_type` which is constructed with an index of an input, or `inarc_index_type` which can be gathered from `input_index_type` using the specific `basic_box` member function. The Bobox framework also provides a helper macro for the assignment of names to inputs.

Currently, the optimizer works only with names of static member functions generated by the helper macro and identifies inputs by these names. In a future implementation, it can identify inputs by indices, but it requires a more complex implementation with an extensive usage of the constant expression evaluation.

5.3.2 Prefetched inputs

For already prefetched inputs, the overridden `init_impl` function is searched. The optimizer searches for `prefetch_envelope` member function calls. It checks function calls whether a callee is the one from the `basic_box`[4] class and collects input names that could be resolved from the first parameter. The first parameter is expected to be a call to the related static member function generated by the helper macro. Actually, a prefetch call is expected to look exactly as the injected prefetch call by the optimizer, see Listing 5.5.

[4]A function with the same name but a different signature can be implemented. Such function hides the base implementation.

```
prefetch_envelope(inputs::left());
```

Listing 5.5: An injected prefetch call for an input called *left*.

5.3.3 Used inputs

Two member functions on a box are searched for a usage of inputs, `sync_body` and `sync_mach_etwas`. These member functions represent the action step. When a member function with such name is found, it is tested whether it overrides the `basic_box` member function.

There are two cases when data from input is considered to be necessary:

1. If there is a call to the `pop_envelope` function on the `basic_box` class. The name of an input is resolved from the first parameter.

2. If there is a helper variable of the type `input_stream<>` for working with a box input and there is a call to any member function on this variable. Listing 5.6 shows a small snippet of the code with the described situation.

Listing 5.6: An example of a *used* input.

```
input_stream<> left(this, input_to_inarc(inputs::left()));
...
if (left.eof())
{
    ...
}
```

5.4 Performance

Actually, the analysis does not traverse whole bodies of functions for every input. Instead of that, it traverses a function body only once, collecting values and using set intersection and join operations on selection statements. For example, on the `if` selection statement it traverses then and else branches collecting values (i.e., names of inputs) and creates the intersection of both sets of names found in these branches.

Therefore, the analysis is very fast. Even though the set intersection operation itself creates a complexity of $n * m^2$, where n is the number of branches created by selection statements and m is the number of box inputs[5], neither of values is expected to be high. The rest of the search algorithm has the linear complexity to the number of nodes in AST.

[5]A set of collected values from a single code branch is sorted before the intersection is created.

5.5 Summary

The only concern about achieved results is the possible big number of false positives when assuming data from an input is necessary as it is described in the second case in the Section 5.3.3. An example in Listing 5.6 shows the situation when input is *probably* necessary only in the single branch of the code, which is not a strong assumption. It was necessary to make such soft assumption in order to make the optimization get the expected result on some tested scenarios. The analysis can be vastly enhanced in this particular part in future.

6. Yield complex method

The Bobox scheduler is a cooperative scheduler, thus it inherits its behavior. The efficiency of programs running in a parallel environment with cooperative scheduling tightly depends on the user code. A task must finish or give up its execution in order to execute a different task on the same CPU. If tasks depend on each other in some way, they should keep balanced execution times as much as possible. Furthermore, too little execution times cause that scheduling exceeds the *real* execution in the CPU consumption[1], big execution times can inhibit parallelism by keeping dependent tasks out of an execution. Big execution times of tasks producing data can also congest framework internal structures.

This optimization method aims to resolve big execution times. The main goal is to detect complex tasks and yield their execution in appropriate places in the code. Listing 6.1 contains the signature of the `basic_box` member functions for such purpose.

Listing 6.1: The signature of the yield execution function.

```
void yield();
```

6.1 Complexity

There are multiple ways to measure the code complexity. Ideally, if we know input data, we run a program with this data and measure its performance. This process is called *profiling*. The quality of optimizations based on profiling is very high, but it requires human resources to analyse data and update the code[2]. There are various techniques used for measurements such as statistical sampling with the hardware support which is very fast, or instrumentation which is more intrusive thus affecting an application performance, but given information is more precise. Instrumentation is useful for applications with a repetitive step with a limit for minimum execution time, or to keep execution times of a step as stable as possible. It helps to find cause of spikes. Such applications are computer games, many sorts of simulations or GUI applications.

Another approach for measuring the complexity is to compile the code and consider the number of generated instructions as the magnitude of the complexity. This assumption is naive and imprecise. The main source of the complexity in most applications comes from loops, repeatedly executed code paths, and this information is not present in such metric. For snippets of a code without a loop, this method is precise enough even if there are multiple execution paths. In bigger code samples, there is probably a bigger amount of loops, thus the code is more complex, but the number of instructions reflects it less and less precise.

[1]The prefetch optimization method described in the previous chapter aims to reduce a number of such executions.

[2]Most popular compilers provide a feature for optimizations based on profiling data, *Profile Guided Optimizations (PGO)* [35], but this approach cannot replace the higher level look on algorithms used in an application provided by a programmer.

The complexity can be also measured based on an indentation in the source code. A bigger maximal indentation usually means more complex code. This approach makes assumptions about source code formatting. Its preciseness tightly depends on these assumptions and how the source code follows them.

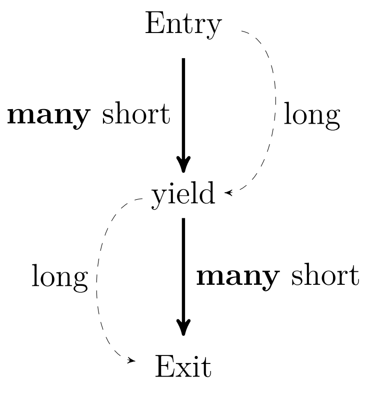
Company programming rules often contain a rule related to the concept called *cyclomatic complexity* [36]. It measures the logical complexity, the number of linearly independent paths through the code. This optimization method aims to reduce a different kind of the code complexity. More code branches decreases readability for a human, but it has almost no effect on runtime.

6.2 Control flow graph

The optimizer has to be able to estimate the complexity of all paths in the control flow graph passing through every graph node representing a code block to find the proper place for a call to the yield member function. The goal is to *cut* long execution paths exceeding some predefined threshold. For simplification, the analyser assumes that all paths have the same probability. Since multiple execution paths often pass through a specific code block represented by a graph node, the insertion of the yield call into this code block cuts all paths passing through this block. In other words, in order to reduce execution time of a single path, such action can cause too little execution times for some other paths. The benefit can be lower than the cost.

Figure 6.1 shows an example of the described situation. The single long execution path is the thinner dashed curved path. Thicker lines represent multiple short execution paths. The yielded block cuts many short paths in order to get rid off the single long path. More short paths mean a lower probability of the long path being taken, thus a lower probability of the optimization benefit. Furthermore, it causes more scheduling so it can even slow down the overall performance.

Figure 6.1: An example of a wrong yield placement into a block shared by multiple short paths and a single long path.



6.2.1 Block complexity

In order to measure complexities of paths in CFG, the optimizer must be able to measure the complexity of the basic construction element of a path, a code block. Only single execution path passes through a code block. Statements in a code block are executed one by one. When a control flow enters a block, if no exception occurs, each statement is executed exactly once until the control flow exits the block. For such code block, the best approach to measure the complexity is an approach similar to measuring the complexity of code by the number of generated instructions, see the second paragraph in Section 6.1.

Code blocks consist of statements. Most statements generate zero or more instructions with constant execution time. Problematic statements are call expressions, because they effectively transfer an execution out of CFG. Their complexity is unknown. The used solution is to estimate their complexity and assign fixed values for different types of call expressions. A block complexity is then a sum of complexities of all statements in this block using values from Table 6.1. The value is searched from the top to the bottom of the table for the first matching row. Trivial, constant and inline call expressions create a subset of call expressions which create a subset of statements. The tool allows a user to change all values by providing a custom configuration. Since trivial call expressions generate no instructions, their default complexity is zero. A compiler evaluates constant call expressions during a compilation and does not generate any instructions for them, thus their default complexity is also zero. The complexity of inlined functions and the rest of call expressions cannot be stated precisely. Even though complexities of such functions vary, their values are very probably in some reasonable sized range. Values for inlined call expressions and the rest of call expressions in Table 6.1 were calculated based on statistics gathered from a code used in benchmarks, see Chapter 8. The function complexity is calculated simply as a number of statements in its body. Appendix C contains complete statistics of measured functions complexities.

Table 6.1: Complexities of statements in a block.

Trivial call expression	
A function body does not generate any instruction, the body is empty.	0
Constant call expression	
A function defined as a constant expression.	0
Inlined call expression	
A function is decided to be inlined by the compiler.	10
Call expression	40
Statement	1

6.2.2 Path complexity

A code block is the basic construction element of a code path. With the definition of the code block complexity, it is possible to define a code path complexity as well. Every CFG constructed in Clang analyzer contains two specially designed empty blocks, i.e., *Entry* and *Exit* blocks. Control flow enters a graph through the *Entry* block and leaves a graph through the *Exit* block. However, if there are loops in CFG, there is an infinite number of paths from *Entry* to *Exit* blocks.

Therefore, loop bodies are evaluated as independent CFG making source CFG acyclic. When a path enters a node with a loop statement as a terminator, this path creates a new path for every path in the loop body. The path that skips a loop body is omitted from the analysis. This path has a very low probability, but it affects results significantly. New paths behave as they skip a loop body, but their complexity is a sum of the source path complexity, block complexity and body path complexity multiplied by a predefined constant from Table 6.2. The optimizer allows users to provide their custom values using a custom configuration. Values in Table 6.2 were calculated based on tool code and its execution on code used in benchmarks, see Chapter 8. Appendix C contains measurements of loop body executions for all `for` loops and all `while` loops. Basically, values in Table 6.2 are averages of an average number of loop body executions.

Table 6.2: Multipliers for loop body complexities.

`for` statement	5
`while` statement	15

Figure 6.2 shows an example of a part of CFG with a `for` loop statement and an `if` selection statement in the body of this loop, see Listing 6.2 for C++ code. Numbers next to edges represent paths complexities and numbers in graph nodes represent blocks complexities. There is a single path with the complexity of 5 entering the node with the `for` statement terminator. The loop body itself contains two different paths with complexities of 2 and 3. Two paths leaving the block with the `for` statement terminator have complexities of 16 and 21. The calculation for those paths is *the `for` loop multiplier * the body path complexity + the entering path complexity + the complexity of the block with the `for` statement terminator*, see Equations (6.1) and (6.2) for paths leaving the `for` statement from Figure 6.2.

$$(5 * 2) + 5 + 1 = 16 \tag{6.1}$$

$$(5 * 3) + 5 + 1 = 21 \tag{6.2}$$

Listing 6.2: A loop statement and a selection statment in a loop body.

```
for (...)
{
    if (/* the complexity is 1 */)
    {
        /* the complexity is 2 */
    }
    else
    {
        /* the complexity is 1 */
    }
}
```

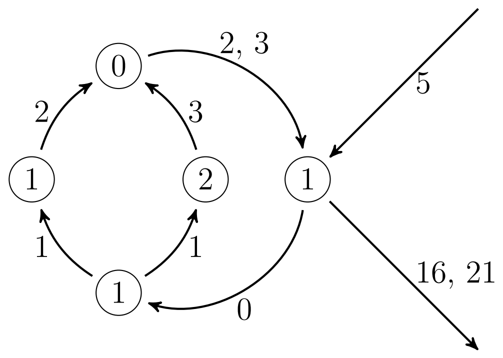


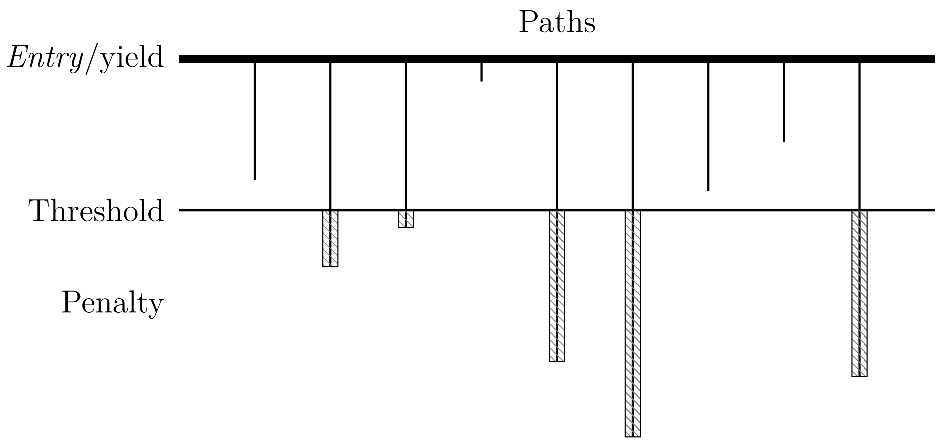
Figure 6.2: A path passing through a node with the `for` statement terminator and a selection statement in its body, see Listing 6.2 for the code example.

6.2.3 Quality of CFG

Briefly, the algorithm to reduce a number of complex paths tries to inject the yield function call into some CFG block to make it *better* than source CFG. Such algorithm needs a value to quantify the CFG quality to compare source and result graphs.

The goal of this optimization method is to reduce a number of complex paths in long-running tasks. For simplification, any path with complexity bigger than some predefined constant is considered to be too complex. Figure 6.3 shows one approach for the CFG quality evaluation. The top horizontal line represents either *Entry* block or a block with the yield call expression, i.e., a block where a control flow enters CFG. Vertical lines represent different paths and line lengths represent paths complexities. The *threshold* horizontal line represents the constant complexity value to distinguish too complex paths. Parts of paths exceeding a threshold are highlighted. Thus, the goal is to minimize the length of highlighted parts of paths. The highlighted part of a path exceeding a threshold is called *penalty.*

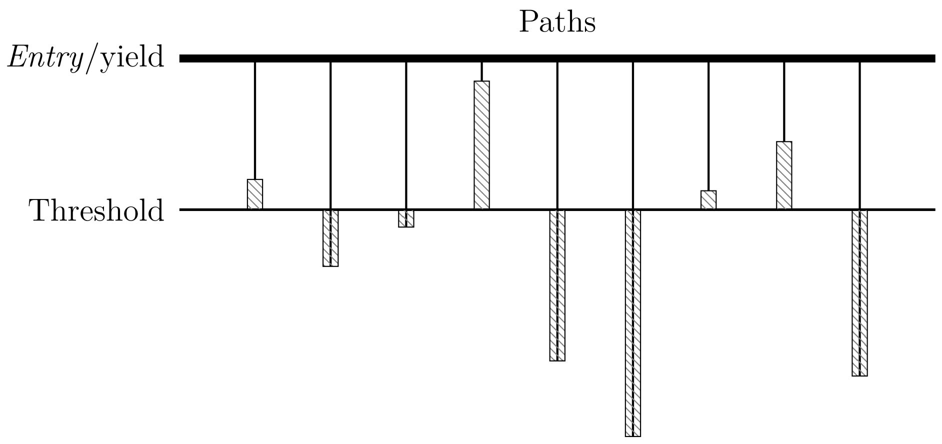
Figure 6.3: The *penalty* approach to evaluate a CFG quality.



When the algorithm to reduce a number of complex paths places a call to the yield function into some block, it cuts one or more paths. A new call to the yield function adds a work for the scheduler and does not remove any work from a task. Thus, a number of new calls to the yield function should be minimized. Unfortunately, there are situations when a placement of the yield function call is equally good for multiple CFG blocks. The simple situation with a single path exceeding a threshold by a single statement is an example. The algorithm can place a call to the yield function into any block passed by this path and it would minimize its penalty to zero. In some edge scenarios, the algorithm could potentially cut only the last statement of the path. It is not the desired behavior since one statement cannot inhibit parallelism significantly. The path should be cut close to its middle or not at all if the yield function call affects other paths negatively.

The solution is to take also other paths than only those too complex into an account in the CFG quality evaluation. Then, the algorithm calculates a sum of distances of path complexities from a threshold value, see Figure 6.4. This approach takes into an account also the disadvantage of placing the yield call expression into some block. The optimizer achieves very good results with such metric of the CFG quality.

Figure 6.4: The *distance from threshold* approach to evaluate the CFG quality.



6.2.4 Additional data structures in optimizer

The analyser needs to keep additional data to CFG when calculating paths complexities. Fortunately, each block in CFG has a unique identifier. Additional data for each block is stored in a map with a block identifier as the key.

Data about every path is stored for every block it passes. Every path has its own unique identifier. Because many paths share their beginnings, information about their complexities from the *Entry* block to a block they pass is shared between these paths in the structure called `path_data_type` with:

- Set of path identifiers.
- Complexity.

Block data consists of:

- Set of `path_data_type` structures.
- A yield state of a block with three different states:

  No There is no yield call expression in this block.

  Planned The optimizer plans to put yield into this block.

  Present Source code already includes the yield call expression.

  Note: Distinguish between *Present* and *Planned* states is to simplify the final code transformation.
- A map of a path identifier as the key and the complexity as a value for blocks with a loop statement terminator. A map holds complexities of loop body paths.

This data is an input for the CFG quality evaluation. The optimization algorithm changes yield states of blocks to the *Planned* state in order to increase the CFG quality.

6.2.5 Optimization algorithm

With a metric for the CFG quality, it is possible to describe the algorithm for the yield complex optimization formally and in more details, see Figure 6.5. The algorithm runs the optimization step on CFG until this optimization step returns CFG with a better quality. The *distance* variable is a sum of distances of paths complexities from a threshold value, a metric described in Section 6.2.3. The algorithm tries to minimize this distance value. The optimization step places zero or one yield into data representing CFG. Data returned from the optimization step is then evaluated and tested whether its quality increased.

Figure 6.5: The algorithm for the yield complex optimization method.

```
1. cfg = Build CFG data.
2. distance = Calculate the quality of cfg.
3. temp_cfg = Run the optimization step on cfg.
4. temp_distance = Calculate the quality of temp_cfg.
5. If temp_distance < distance then
   5.1. distance = temp_distance.
   5.2. Swap cfg with temp_cfg.
   5.3. Continue in the step 3.
6. Else finish, cfg is optimized.
```

Complexities calculation and quality evaluation

The optimizer uses the depth-first search algorithm to analyse CFG and calculate complexities. The analysis has to handle yield calls in blocks, which are already in the code or are planned for an insertion by the optimizer. Inputs of the analysis are CFG and data structures described in Section 6.2.4. The CFG quality evaluation gets paths and their complexities as an input and calculates a value representing the CFG quality described in Section 6.2.3.

Optimization step

The only goal of the optimization step is to decrease the distance value representing the quality of CFG. The optimizer uses brute force to achieve the goal. Firstly, it collects all blocks where at least one path ends, i.e., the *Exit* block and blocks with the *Planned* or the *Present* yield state. Then, it processes every block with at least one path with complexity higher than the threshold value and calculates what happens if the yield call expression is placed into that block. A block with the best outcome has its yield state set to *Planned.*

The complexity of such algorithm tightly depends on the complexity of the user code and on the structure of its CFG. Every block is visited twice in each optimization step. All paths complexities are recalculated[3] for every block with at least one path with complexity higher than the threshold value. However, the next optimization step is executed only if the previous optimization step has changed the yield state of one block into the *Planned* state. Thus, the optimization step has increased a number of paths, but greatly decreased a number of blocks with too complex paths.

6.2.6 Default threshold

A presence of loops reflects the code complexity the most. However, a loop body has to be appropriately complex to make the whole loop complex. The only non-trivial metric used in this optimization method for the complexity is again a loop. If both loops are `for` loops, then based on values from Tables 6.1 and 6.2, the inner loop body must be more complex than a single inlined call expression. Otherwise, the inner loop would have the complexity of 50, approximately as big as a single non-inlined call expression with the complexity of 40. I have decided for two non-inlined call expressions since only one is the minimal case and five of them is basically another inner loop with one call expression.

Using values from Tables 6.1 and 6.2, the default value for threshold is calculated as the execution of five inner `for` loops with two non-inlined, non-trivial, non-constant call expressions, see Equation 6.3.

$$5 * 5 * 2 * 40 = 2000 \tag{6.3}$$

6.2.7 Code injection

The last step of the optimization process is the injection of the yield call expression to blocks with the *Planned* yield state. A block with such yield state can be

[3]A recalculation of a path complexity means one subtraction from its complexity.

empty, it can contain a single statement in a condition expression of a selection statement, the right-hand side expression in a binary expression, the else branch in a conditional expression, or any other language structure where the C++ language grammar does not allow to chain statements.

The easy solution is to find a compound statement, where the injection of the yield call expression is as good as the injection directly to the chosen block. The injection of the yield call expression into a compound statement is then simple and safe. For reminder, the prefetch optimization method already does inject a code into compound statements.

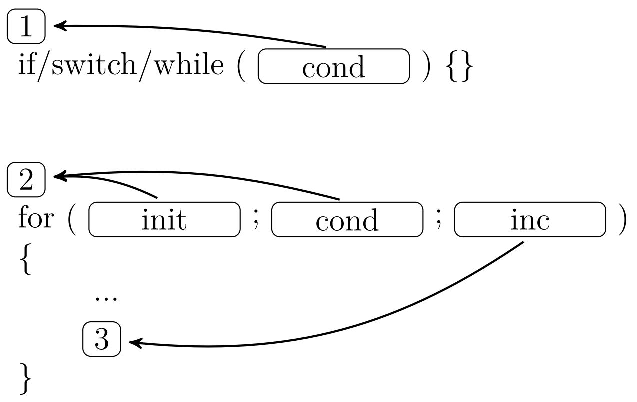
Firstly, the optimization method collects all compound statements in the function body compound statement[4]. Then, the optimizer collects all blocks with the *Planned* yield state. For every collected block, the optimizer checks all compound statements whether the relation of the collected block and the compound statement matches any of special cases, see Figure 6.6.

1. If the block with the *Planned* yield state is the condition expression of the `if`, `switch` or `while` statement that is a child statement of the compound statement, the yield call expression is injected just before the `if`, `switch` or `while` statement.

2. If the block with the *Planned* yield state is the initialization statement or the condition expression of the `for` statement that is a child of the compound statement, the yield call expression is injected just before the `for` statement.

3. If the block with the *Planned* yield state is the incremental expression of the `for` statement and the compound statement is its body, the yield call expression is injected as the last statement of the compound statement.

Figure 6.6: The injection of the yield call for some special cases of statements.



6.3 Further improvements

The current state of the optimization method can be still improved. Some indications about further improvements were already mentioned, e.g., producers

[4]The function body compound statement is included in this set

can congest internal framework structures in the introduction of this chapter, or different path probabilities in the introduction of Section 6.2.

6.3.1 Runtime checks

If auto-vectorization back-end optimizers cannot prove that operations in a loop body do not overlap in the compilation process, they generate runtime checks and both versions of a loop, original and vectorized. Although there is nothing to *prove* mentioned in the yield complex optimization method description, there is a lot of estimations such as call expression complexities or loop body multipliers. It would be possible to handle suspicious cases more precisely using runtime checks.

6.3.2 Probabilities

The optimization method considers that all paths in CFG have the same probability. It is a simplification. For example, functions often contain multiple checks of their arguments on the beginning of their bodies, but the analysis can assume that these branches will not be taken in majority of function calls because inputs are expected to be correct. What probabilities should be assigned to paths is a very complex task beyond a scope of my thesis. Developers of branch predictors on modern processors confront the similar task [37]. Some ideas for a branch prediction could be reused for the static analysis, but most mechanisms used for predictions are based on runtime information.

6.3.3 Identify producers

There are multiple drawbacks of long-running tasks mentioned in the introduction of this chapter. One of them is the possible congestion of framework internal structures. The analysis can assign more *weight* to paths that produce data for other tasks in order to ease their yield. Loops producing data deserves more recognition than loops performing calculations.

6.3.4 Deep analysis

Probably the simplest case of an improvement for the optimizer is the deeper analysis of statements in CFG blocks. Complexities of call expressions are estimated, but some of them can be calculated more precise. All categories of call expressions can be analysed deeper if the analysis has an access to the body of a callee. It would be unbearable to count in CFG of the function definition, but some heuristic based on a callee definition can be helpful. The possible result of such heuristic can be in form of the depth of the most nested loop.

7. Optimizer

The tool is implemented on top of Clang using LibTooling and AST matchers libraries, see Section 4.6. The current implementation provides two different methods for the optimization of code using the Bobox framework. However, the tool design allows an easy implementation and an integration of new optimization methods.

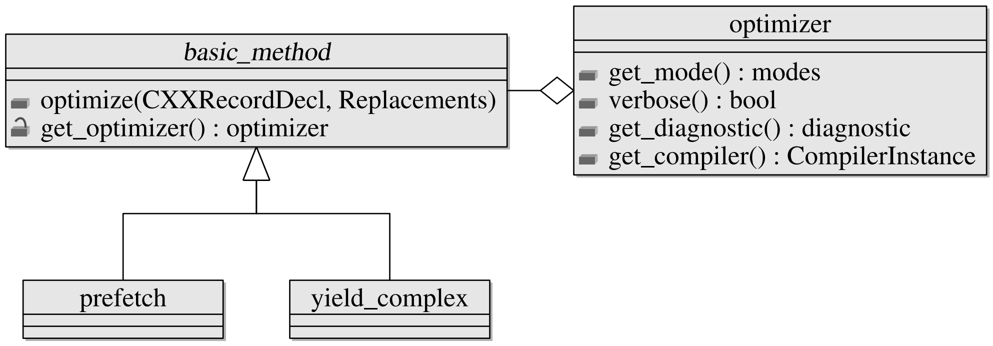
The optimizer is able to perform source-to-source transformations only on the text level. It is designed to be used for the diagnostic purpose or in the build process.

7.1 Design

User code of a Bobox task is placed in a class derived from the `basic_box` class. Before the optimization starts, the optimizer looks up all boxes defined in source code. The AST matchers interface fits this purpose the best. The central `optimizer` class is the callback from the AST matchers library. The class also provides all optimization methods with various information. The next step is to distribute handles to boxes to all optimization methods and providing them with tool runtime data by providing a handle to the `optimizer` object.

Based on the command line parameter that defines the optimization level[1], the `optimizer` object allocates objects representing optimization methods. The `optimizer` object is responsible for their lifetime. Optimization methods have to implement a simple interface, which accepts a handle to the AST node representing a box and handle to the `Replacements` object provided by the AST matchers library, see Figure 7.1. The `optimizer` object also injects a handle to itself into every instantiated method so methods can access tool runtime data.

Figure 7.1: The class diagram for the optimizer core.



optimizer
get_mode() : modes
verbose() : bool
get_diagnostic() : diagnostic
get_compiler() : CompilerInstance
basic_method
optimize(CXXRecordDecl, Replacements)
get_optimizer() : optimizer
prefetch
yield_complex

The implementation and the integration of a new optimization method consists from two steps:

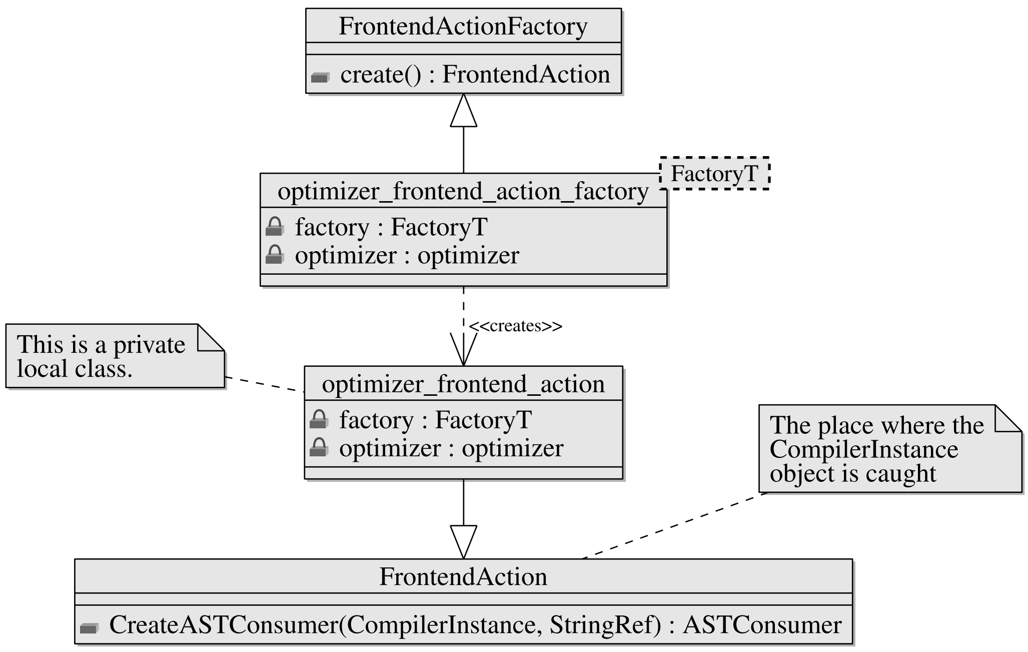
1. Implement the `basic_method` interface, which consists from the single member function to invoke a box optimization.

[1]Similar to what modern C++ compilers support, e.g., *GCC* and *Clang/LLVM* `-Ox` command line option and *Microsoft Visual C++* `/Ox` command line option.

2. Register the method factory function in the `method_factory` class. The method factory function has to create the optimization method object giving up its ownership.

The AST matchers interface does not provide the access to Clang compiler internals. It provides a developer with a handle to the matched node, `ASTContext` and `SourceManager`. It was necessary to implement own wrappers for the front-end action and the AST consumer to catch the handle to the compiler main object of the `CompilerInstance` type, see Figure 7.2. The other approach to get the access to compiler internals would be to directly change Clang source code. This approach would introduce significant problems. In such case, tool source code has to be distributed with modified Clang source code. Also, resolving potential changes in the Clang tooling interface semantic is easier in separate code base rather than merging or updating Clang code base.

Figure 7.2: The class diagram for wrapping of Clang tooling API.


FrontendActionFactory
create() : FrontendAction
optimizer_frontend_action_factory
FactoryT
factory : FactoryT
optimizer : optimizer
<<creates>>
This is a private local class.
optimizer_frontend_action
factory : FactoryT
optimizer : optimizer
The place where the CompilerInstance object is caught
FrontendAction
CreateASTConsumer(CompilerInstance, StringRef) : ASTConsumer

7.2 Working modes

The tool is primarily supposed to be used as the front-end optimizer when it is quietly executed in a build process. Yet, the optimizer can operate in another two modes, diagnostic and interactive. Both modes differ in a verbosity from the mode used in a build process. The tool in these modes outputs reasoning behind the optimization process.

The tool supports three different modes. The text in bold is the command line argument used to switch to the desired mode.

-build The tool runs quietly, transforming source code. The only output is the Clang compiler diagnostic output. There is a rationale behind the

Clang compiler diagnostic output in the tool. If there is a compile error, a user should be able to see why the optimizer does not work. Furthermore, developers should not ignore compile errors and warnings. If developers do not want the compiler diagnostic output, they can filter it out.

Even though it is not necessary to make transformed code look pretty for a human eye because transformed code is supposed to be immediately processed by the compiler, this mode injects code with the correct indentation and line endings.

-diagnostic The diagnostic mode is a verbose mode that does not perform any code transformations. The optimizer outputs problematic parts of user code and rationale behind suggestions. The output also contains pointers to highlight the most important part of the printed code snippet. The diagnostic output is similar to the Clang diagnostic output. However, it is implemented separately.

The rationale behind implementing such mode is that programmers still hesitate to use code transformation tools for C++ apart from formatters and simple refactoring tools. Unless the optimizer output is not too verbose and programmer can process it, it is safer to allocate human resources for the optimization process. A programmer can resolve optimizer suggestions directly in code base. Then, it is not necessary to use the optimizer in a build process, thus making the process faster.

Another reason for this mode is that the tool can be used in environments where source code is read-only. The *Perforce* [38] revision control system is such an example. Perforce does not allow a programmer to edit source code until he marks it as *checked-out*. Checking-out whole code base for editing a couple of files is performance demanding for the server side of the revision control system.

-interactive This mode is equivalent to the diagnostic mode with one additional feature. Each optimizer suggestion comes with yes/no type of a question. A programmer answers whether he desires to apply the transformation on code immediately.

The rationale behind this mode is to make processing of the optimizer diagnostic by a programmer faster. If a programmer sees that a suggestion is relevant, he does not need to switch to another environment to write what he already sees on a screen.

The decision granularity is on the level of optimizer suggestions. The single optimization method may produce multiple suggestions for a single box. A programmer can pick which of them will be applied.

7.2.1 Optimizer output

The tool tries to emulate the Clang diagnostic output as much as possible. The output is based on pointing out specific parts of code snippets, see Listing 7.1. Tool diagnostic code functions receive a location in source code together with a text message and a type of a message. There are three different types of messages.

info General information about source code.

optimization A general message about the optimization process.

suggestion A suggested transformation to source code.

Listing 7.1: An example of the tool diagnostic output.

```
boxes.hpp:60:32: info: missing prefetch for input declared here:
BOBOX_BOX_INPUTS_LIST(left,0, right,1);
                              ^~~~~
```

The tool diagnostic is aware of macro expansions, see Listing 7.1. Yet, it does not show the whole stack of the macro expansion as the Clang diagnostic does. It outputs spelling locations only. Outputs for both optimization methods are more precisely described in Appendix B.

7.3 Coding style detection

The optimizer injects new code the way it tries to follow a coding style as much as possible. An indentation is probably the most visible property of a coding style. Many indent styles have evolved over time. The currently implemented optimization methods try to follow the style of whitespace characters on the beginning of lines only.

The coding style detection algorithm runs over a class definition. Information from a whole memory buffer that represents a whole translation unit would be misleading since a translation unit can consist of many files from different libraries. A class definition is the big enough example to detect the coding style. The tool also detects the style of line endings, whether it is *line feed (LF)* only, or it is paired with *carriage return (CR)*.

The algorithm to detect the indentation processes the memory buffer of a box definition from the start location of a class to its end location. The algorithm can be described by the following seven steps:

1. Set the empty indentation as the last line indentation.

2. If the algorithm has not yet reached the end location of a class, continue in the next step, otherwise continue in the step 7.

3. Find the first non-whitespace character.

4. If the current line contains only whitespace characters or it is a comment, move to the next line and continue in the step 2.

5. Increase occurrences count for the difference between the last remembered indentation and the current line indentation.

6. Remember the current line indentation, move to the next line and continue in the step 2.

7. Pick the most occurred difference. If there is no clear winner, pick tabs.

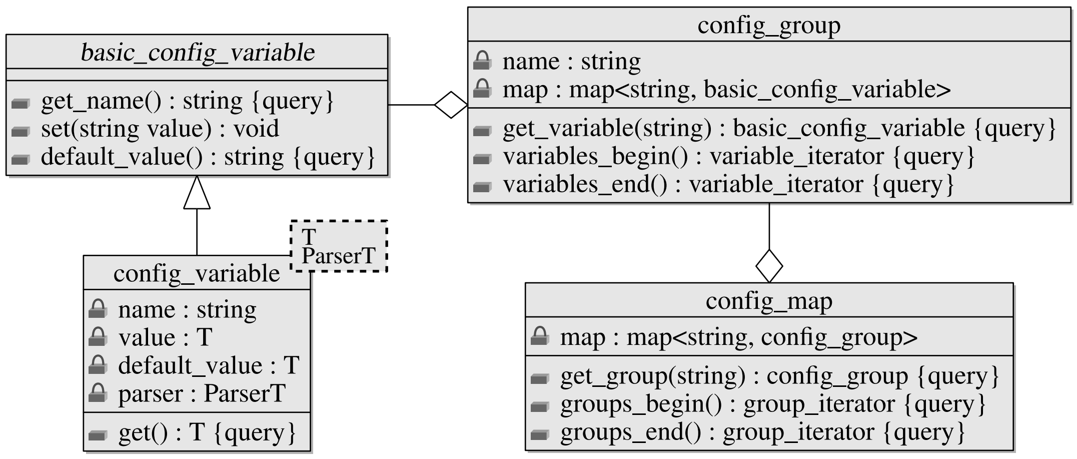
The similar algorithm is used to detect an indentation of member function declarations in a class definition. The algorithm remembers whitespace characters on the beginning of lines with a member function declaration or definition. It picks the one with the most occurrences as the result.

7.4 Configuration

The yield complex optimization method tightly depends on multiple constants. Their values affect the quality of the result. Also results of the prefetch optimization method can vary if the method searches for the used inputs in a loop body or not.

Clang/LLVM code base does not provide any configuration facilities. It was necessary to write it from scratch[2]. Thus, requirements are accordingly low. At minimum, it is necessary to store values of different types paired with a name used as the key. Names must be unique in some scope. Every configuration *variable* has to be assigned to a configuration *group*. Every configuration group has a name unique in the global scope.

Figure 7.3: The class diagram for configuration code.


basic_config_variable
get_name() : string {query}
set(string value) : void
default_value() : string {query}
config_group
name : string
map : map<string, basic_config_variable>
get_variable(string) : basic_config_variable {query}
variables_begin() : variable_iterator {query}
variables_end() : variable_iterator {query}
T
ParserT
config_variable
name : string
value : T
default_value : T
parser : ParserT
get() : T {query}
config_map
map : map<string, config_group>
get_group(string) : config_group {query}
groups_begin() : group_iterator {query}
groups_end() : group_iterator {query}

The class diagram on Figure 7.3 shows the hierarchy of classes in configuration code. The class hierarchy also defines the scope of the uniqueness for configuration elements, i.e., a configuration variable has to have a unique name in the configuration group scope. The `config_map` class uses the Meyers[3] singleton pattern. The class is used as the gateway to all configuration groups and variables.

Four types of a line can appear in the configuration file. Every type of a line is defined by a regular expression.

1. A line representing a configuration group.
   `\[[a-zA-Z0-9_ ]+\]` e.g. `[group name]`

[2]LLVM CommandLine 2.0 library served as the inspiration for the design because of similarities with the goal of the configuration library.

[3]Drawbacks of the Meyers singleton are not present in this usage.

2. A line representing a configuration variable.
   `[a-zA-Z0-9_]+\s*:\s*.*` e.g. `variable: value`

3. An empty line or line that consists from whitespace characters only.
   `\s*`

4. A comment where the first non-whitespace character is `#`.
   `\s*#.*`

Every configuration variable has to be defined with a default value. The tool is able to generate a configuration file with default values.

8. Results

Both optimization methods described in previous chapters should optimize code for the Bobox framework to some extent. This chapter covers performance evaluation of optimized and unoptimized code for both methods. However, each method is evaluated separately. There is a specific use case for each method, when it should stand out.

8.1 Prefetch method

The goal of this optimization method is to reduce a number of short-running tasks. Specifically, tasks that are scheduled even though they do not have all input data for a meaningful execution. The gain in a speedup with this optimization method is the scheduling overhead. It only matters how often this situation occurs in user code.

In order to measure the scheduling overhead, it is necessary to maximize the number of wrongly scheduled tasks. A wrongly scheduled task has to have at least two inputs and there has to be some delay between getting data to its inputs. A bigger delay increases the possibility of the task being scheduled without all necessary data.

Furthermore, a parallel environment is not necessary in order to measure the scheduling overhead. It is harder to achieve the situation when a wrong scheduling matters with more logical threads. Wrong scheduling on free logical threads does not affect the overall performance. Thus, measurements are done on a single thread.

8.1.1 Model

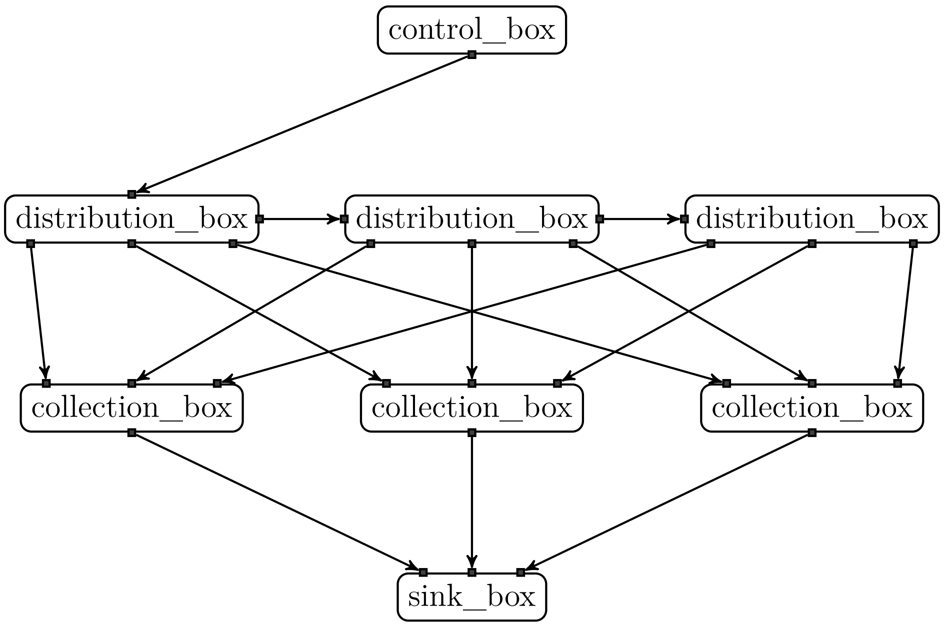
There has to be a box with more than one input for the wrong scheduling and there can be more such boxes for a bigger effect. Now, it is necessary to create some mechanism to maximize the number of situations when a box is scheduled wrongly. This mechanism consist from a chain of boxes where each one creates input data for one specific input on all boxes that are supposed to be scheduled wrongly. All boxes from this chain, apart from the last box, then trigger scheduling of the next box in the chain that will create data for a different input on wrongly scheduled boxes.

Figure 8.1 shows the model used in measurements. However, the number of distribution and collection boxes as well as the number of their inputs and outputs may vary. The figure shows only a *pattern* used in the model construction. If all collection boxes prefetch only the first input or no input at all, the expected scheduling in the execution of a model instance constructed from the model in Figure 8.1 is:

1. control_box
2. distribution_box
3. Three times collection_box - pop data from the first input and wait

4. distribution_box
5. Three times collection_box - pop data from the second input and wait
6. distribution_box
7. Three times collection_box - pop data from the last input and produce an output
8. sink_box

Figure 8.1: The model used in measurements of the prefetch method optimization with three distribution and three collection boxes.



If all collection boxes prefetch their all input data, step 3 and step 5 are skipped. These prefetch calls can save up to six schedules when boxes do nothing meaningful.

8.1.2 Benchmarks

There are two boxes in the model without any description. The control box repeatedly produces data for distribution boxes. It yields each time after data is produced. The sink box is there for a debugging purpose. All boxes do minimal work apart from a communication with the Bobox framework to maximize the impact of the scheduling.

Figure 8.2 shows results for the model with ten distribution and ten collection boxes. The dashed line shows times for not optimized code, the solid line shows times for optimized code. X-axis represents the number of iterations in the control box. Y-axis represents execution times.

Even though the graph does not show it well, optimized code is very slightly faster in most measurements. However, the gain in a speedup is negligible even

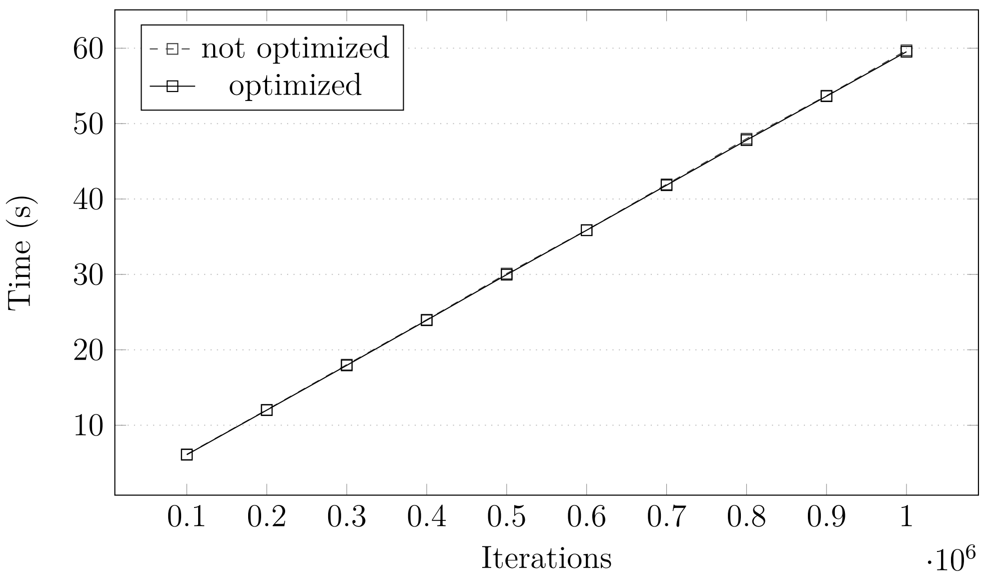


Figure 8.2: Benchmarks for the prefetch optimization method with ten distribution and ten collection boxes.

for a bigger number of iterations or a bigger number of collection and distribution boxes. Furthermore, the gain in a speedup is approximately the same for a bigger number of iterations as for a lower number of iterations. The speedup appears to be constant for measured results. These results do not correspond with the expected behavior.

8.1.3 Optimizer tweak

It is necessary to look more into details of the presented model in Figure 8.1 and the Bobox framework implementation to understand why there is a negligible speedup from unoptimized to optimized code. Both distribution and collection types of boxes are stateless. The Bobox framework allocates some objects of stateless boxes, initializes them by calling the `init_impl` member function and reuses those allocated objects all over again. But, `init_impl` is not called again when object is reused. Thus, prefetch calls added by this optimization method do not help anymore when the framework reuses boxes.

The prefetch optimization method is enhanced to add all prefetch calls for all used inputs to the end of the box execution member function body. This option is configurable and turned off by default. Figure 8.3 shows results with this optimization tweak turned on compared to not optimized code. The improvement in a speedup is clearly visible.

8.1.4 Conclusion

The Bobox scheduler is greatly optimized. Even in the presented scenario with approximately 10 boxes scheduled 9 times out of 10 wrongly, the speedup with 1 million iterations is only 5,767 seconds from 59,718 seconds to 53,951 seconds. Furthermore, tests were measured on a single logical thread. It is much harder to achieve this scenario with multiple logical threads. The scenario itself is artificial

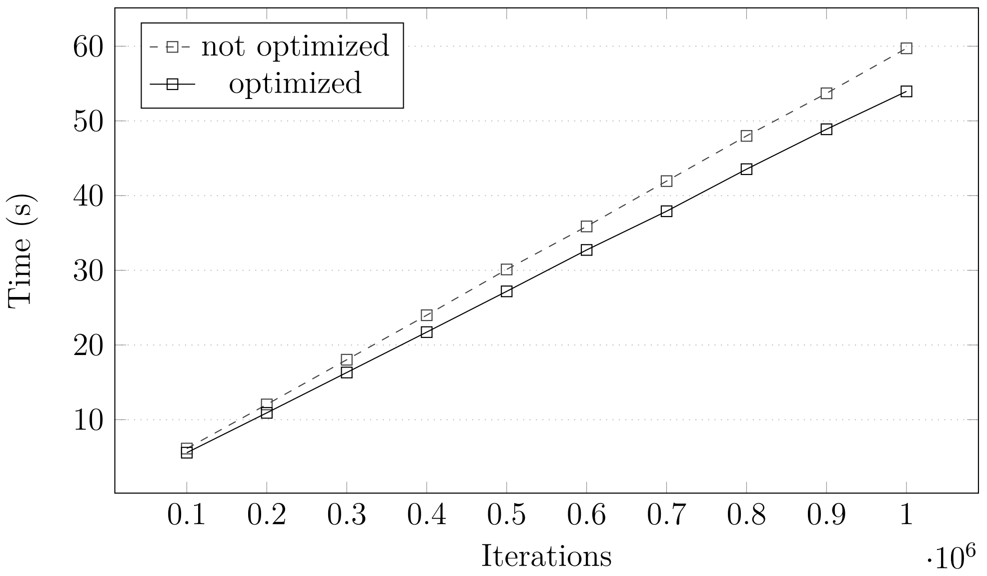


Figure 8.3: Benchmarks for the prefetch optimization method with ten distribution and ten collection boxes and the *attach to execution body* tweak turned on.

to showcase the optimization potential. On the other hand, there are software products where every second matters and there is no reason to turn off this optimization.

However, there is still one factor that is not tested well in the presented scenario, cache usage. Since all boxes do not work with data excessively, their cache usage is minimal. Most data can be probably held in cache the whole execution time. On the other hand, a scheduled box without all input data probably immediately reschedules. Thus, its impact on a cache is negligible anyway.

8.2 Yield complex method

The goal of this optimization method is to reduce a number of long-running tasks. Such tasks can inhibit a parallel execution, thus vastly increase execution times. On the other hand, long-running tasks are more cache friendly, but the speedup from a parallelism should be much bigger than the one from a better cache usage.

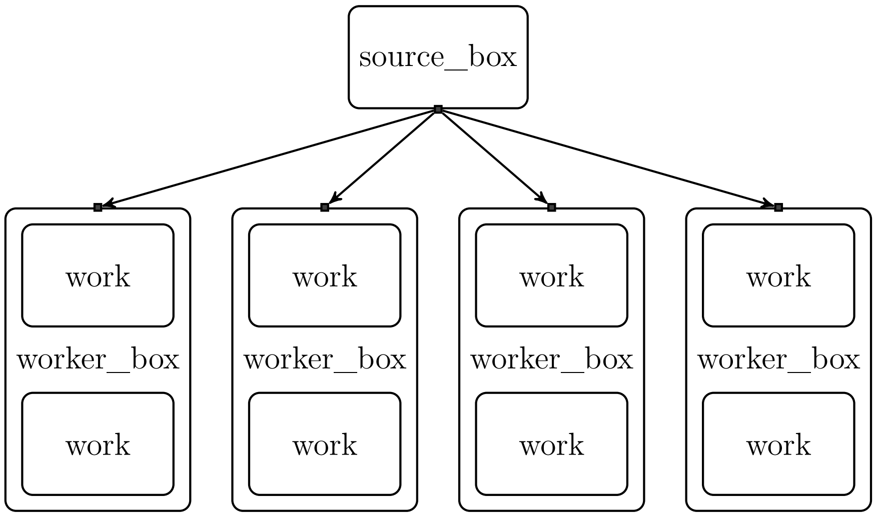
8.2.1 Model

Measurements are done on a model with one important task. This task is essential for a parallel execution and it should run as often as possible. But, there are also long-running tasks which prevent scheduling of this essential task, thus they inhibit parallelism.

Figure 8.4 shows an example of a model with one important task. The source box is a box that computes data for a parallel processing and distributes this data to worker boxes. Worker boxes process data in parallel. If there is the same number of worker boxes as the number of logical threads, there is a big possibility that the scheduler immediately schedules all worker boxes to all logical threads.

The worker box is the long-running task. Thus, all worker boxes keep the source box out of an execution. This source box creates the bottleneck for a parallel execution.

Figure 8.4: The model used in measurements of the yield optimization method with four logical threads.



If worker boxes finish approximately at the same time, the source box is scheduled and until it finishes, nothing runs in parallel. But, if one worker box yields its execution and let the source box to run, data for the next bunch of parallel worker boxes is ready sooner. Even thought the particular worker box ends later, it does not inhibit parallelism as there are always data for other workers. A bigger task granularity greatly helps in this particular model example.

8.2.2 Benchmarks

Measurements were done on a parallel environment with eight logical threads. Therefore, the model consists of eight stateless worker boxes and single source box. The worker box *calculates* for ten seconds and the source box *calculates* for five seconds. There would be different overall times measured for different times of a boxes execution, but the ratio between optimized and unoptimized code is expected to be the same. This optimization method causes the worker box to yield after five seconds. Figure 8.5 shows measured results.

But, measured results in Figure 8.5 are a bit different from the expected outcome. If the source box is the bottleneck of an execution, it should inhibit the parallel execution for five seconds each iteration. So the speedup for ten iterations should be approximately fifty seconds. The measured result is approximately twenty seconds.

The problem is the implementation of the source box for these particular measurements. The source box calculates something for five seconds, then it sends envelopes to all worker boxes and it yields immediately. But, the Bobox scheduler does not schedule all worker boxes immediately after the source box yields. The source box is often scheduled with the bunch of worker boxes and

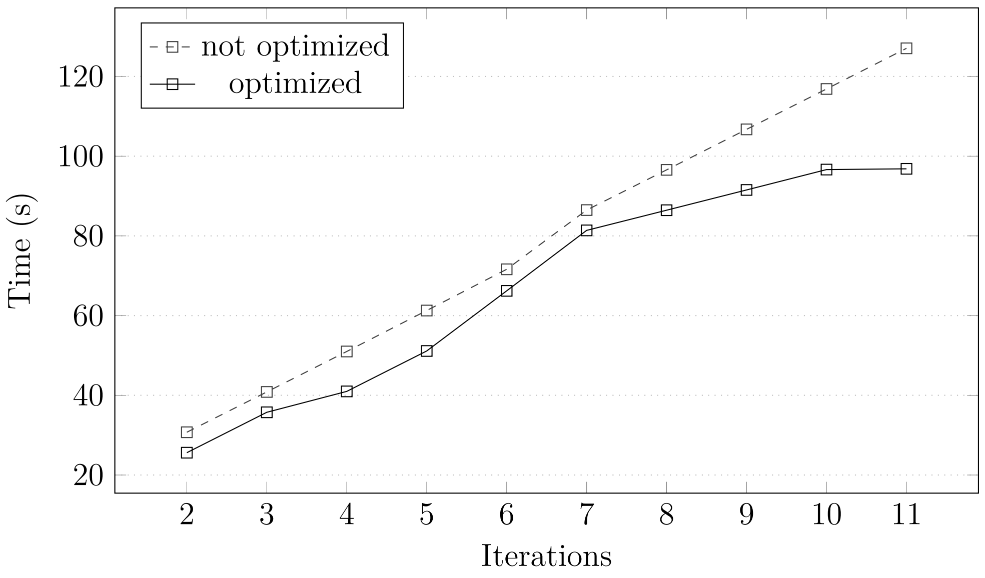
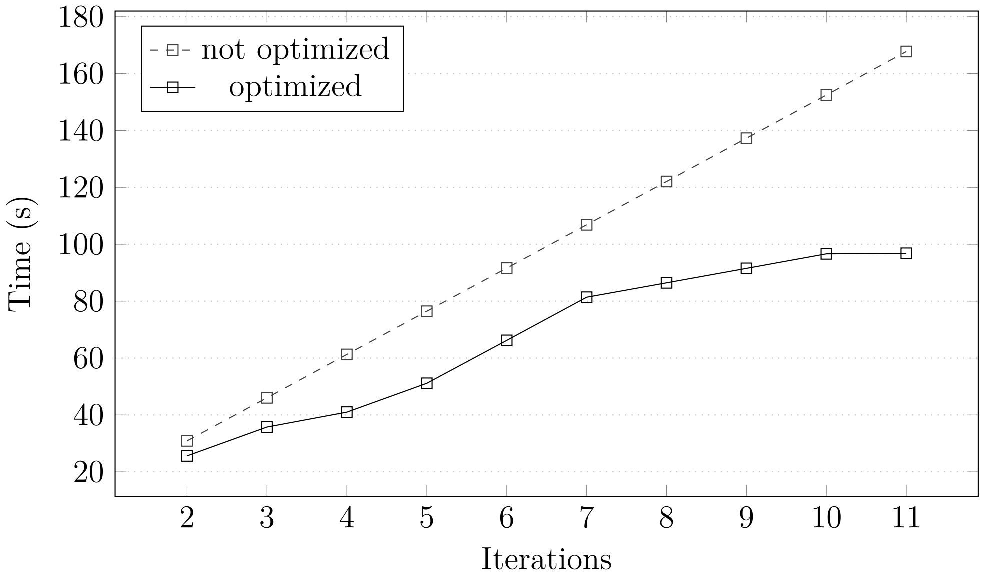


Figure 8.5: Benchmarks for the yield optimization method with eight logical threads and eight worker boxes, the naive implementation.

delays an execution of one of them. However, there is still a significant speedup in the execution.

The source box has to do some additional work before it yields to achieve the described behavior. It gives time to the Bobox scheduler to schedule seven worker boxes and the yield in the source box schedules the last worker box. Figure 8.6 shows measured results with this described implementation and the speedup approximately matches expected numbers.

Figure 8.6: Benchmarks for the yield optimization method with eight logical threads and eight worker boxes.



8.2.3 Conclusion

The presented model lacks a good parallel design in the first place and a more skilled programmer would not probably write such code. But, this model can be *hidden* in more complex models.

Long-running tasks can occur in a longer development process, when programmers extend a task to do more and more work with increasing requirements for application features. There may be various reasons such as a lack of time or even laziness to write code directly to an existing task. A programmer can also develop a long-running task because the work done by this task is monolithic and it is natural to write such code in a single function. This programmer may not realize consequences of such code to parallelism and forget to yield its execution. There are multiple ways how long-running tasks can occur in a code. This paragraph mentions only some of them.

The bigger granularity can greatly speedup an application execution. Furthermore, results of the prefetch optimization method show that even excessively short-running tasks that run excessively often affect execution times only very little. This optimization method can definitely speedup an application with very low risks.

9. Conclusion

The Bobox project is a framework for task-based parallel computing. The task-based approach relieves a programmer from various issues necessary to handle in the thread-based approach. It is definitely an approach that will be primarily used in the future. Nowadays, most parallel computing software has already made the transition to the task-based implementation. The rest is encouraged to do so.

However, parallel computing is not for free even in the task-based environment. The environment handles programming issues related to parallel computing and it does bring some overhead. This overhead concentrates in times when a task starts and finishes its execution. Thus, tasks should run for enough time to make this overhead negligible. The prefetch optimization method tries to reduce special cases of short-running tasks. Running a task without necessary input data causes this task to finish almost immediately. This method detects such tasks and informs the Bobox framework about their input requirements. The analysis makes some assumptions, which can result in false positives. However, if a task accesses input data using some of expected ways, this method detects it. The scheduling mechanism in the Bobox framework is greatly optimized. The gain in a speed is not overwhelming even in cases where this optimization method excessively reduces an amount of unnecessary scheduling. On the other hand, there is no reason to refuse any gain in an application speed.

The Bobox cooperative scheduling of tasks brings another responsibility on a task developer. A task should not run for a long time. If a task produces data for other tasks, it can inhibit the parallel execution, because other tasks wait until this task finishes. It can also congest internal framework structures. Such task should yield its execution after some time. The yield complex optimization method detects complex tasks and injects yields to appropriate places in code. Measuring a complexity by the static analysis is a hard task since the real complexity tightly depends on input data. The analysis has to make many assumptions about code, thus it results in more false positives. Nonetheless, the yield operation is not harmful to a performance. On one side, it can greatly help an application performance, on the other side, it will not hurt its performance unless its excessive usage. The method analyses general C++ code and the only related operation to the Bobox framework is the yield call. Thus, anyone who is interested in the code complexity can reuse the algorithm. This optimization method shows a great potential in the measured use case. Even though, the use case is artificial, it can appear in some context in more complex models. Furthermore, there are more use cases where a bigger task granularity helps.

The only concern for both methods is the possible high ratio of false positives. Both methods make assumptions about code, e.g., a loop body is executed at least once. Therefore, the prefetch method can introduce prefetch of some input data even though a task is able to do some work without it. The yield complex method can place yield after a loop whose body is not executed, thus introduce a short-running task. A user of the tool should know about these drawbacks and use it carefully. However, the tool provides diagnostic and interactive modes, which are very useful and harmless.

9.1 Future work

The tool is designed for further enhancements like adding new optimization methods or improving the implementation of current ones. Some ideas for future improvements of the yield optimization method are mentioned in Section 6.3.

Furthermore, the tool is implemented as a standalone tool, but because it is implemented on top of LibTooling and AST matchers libraries, it can be easily implemented as a Clang plugin, sharing a big part of codebase. There are also possibilities of improvements in the internal tool implementation. The tool currently uses the `Rewriter` class for source-to-source transformation, but the recommended approach is to use the `TreeTransform` class. In the tool usage, diagnostic messages can be more verbose. Even though data from an analysis is available in the tool, there is no simple way to display them to a user. One issue was also encountered during the development. The tool cannot run in parallel, which can be major issue when using the tool in a build environment.

However, when the tool was designed, most of mentioned issues and missing features were already known so they are included in the design. During the development, the main focus was on the implementation of the optimization methods and used algorithms. The future work consist mainly from fixing issues, improving the implementation and the interaction with a user.

References

[1] Bednárek, David. – Dokulil, Jiří. – Yaghob, Jakub. – Zavoral, Filip. *Parallelization Framework for Data Processing.* Advances in Information Technology and Applied Computing. ISSN: 2251-3418, 2012.

[2] Bednárek, David. – Dokulil, Jiří. – Yaghob, Jakub. – Zavoral, Filip. *Data-Flow Awareness in Parallel Data Processing*, in Intelligent Distributed Computing IV, Springer. ISBN: 978-3-642-32523-6, 2012.

[3] Dokulil, Jiří. – Bednárek, David. – Yaghob, Jakub. *The Bobox Project: Parallelization Framework and Server for Data Processing.* Technical report no. 2011/1. Department of Software Engineering, 2011.

[4] Bednárek, David. – Dokulil, Jiří. – Yaghob, Jakub. – Zavoral, Filip. *Bobox: Parallelization Framework for Data Processing.* Accepted for publication in Advances in Information Technology and Applied Computing. ISSN: 2251-3418, pp. 189-194, 2012.

[5] *Bobox* [online]. A highly parallel framework for data processing. [cit. 2014-02-28] URL: <`http://www.ksi.mff.cuni.cz/cs/sw/bobox.html`>.

[6] Falt, Zbyněk. – Bednárek, David. – Kruliš, Martin. – Yaghob, Jakub. – Zavoral, Filip. *Bobolang - A Language for Parallel Streaming Applications.* In Proceedings of the 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing, Vancouver, ACM. ISBN: 978-1-4503-2749-7, pp. 311-314, 2014.

[7] ISO/IEC 14882. *Working Draft, Standard for Programming Language C++.* JTC1/SC22/WG21. Document number: N3797. Date: 2013-10-13.

[8] *Sparse FAQ* [repository]. 66b24573e9cb5eaa0c41dc4164f81f3b83b9cb41:FAQ. URL: <`https://git.kernel.org/pub/scm/devel/sparse/sparse.git`>.

[9] Hall, Peter. *Crysis 2 Multiplayer : A Programmer's Postmortem* [online]. GDC Europe 2011. [cit. 2014-02-26]. URL: <`http://www.gdcvault.com/play/1014887/Crysis-2-Multiplayer-A-Programmer`>.

[10] Zaks, Anna. – Rose, Jordan. *How to write a checker in 24 hours* [online]. Clang Static Analyzer. [cit. 2014-02-26]. URL: <`http://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf`>.

[11] *Clang Static Analyzer* [online]. Development manual. [cit. 2014-02-26]. URL: <`http://clang-analyzer.llvm.org/checker_dev_manual.html`>.

[12] *Design: clang-format* [online]. Design document. [cit. 2014-02-26]. URL: <`https://docs.google.com/document/d/1gpckL2U_6QuU9YW2L1ABsc4Fcogn5UngKk7fE5dDOoA`>.

[13] *Cppcheck doxumentation* [online]. Generated by Doxygen. [cit. 2014-02-26]. URL: <`http://cppcheck.sourceforge.net/devinfo/doxyoutput`>.

[14] *"Clang": a C language family frontend for LLVM* [online]. Clang homepage [cit. 2014-03-25]. URL: <`http://clang.llvm.org/`>.

[15] *Clang 3.5 documentation* [online]. [cit. 2014-02-28]. URL: <`http://clang.llvm.org/docs/index.html`>.

[16] *"Clang" CFE Internals Manual* [online]. Clang 3.5 Documentation. [cit. 2014-02-27]. URL: <`http://clang.llvm.org/docs/InternalsManual.html`>.

[17] WIKIPEDIA. *Abstract syntax tree — Wikipedia, The Free Encyclopedia* [online]. [cit. 2014-02-28]. URL: <`http://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=595980085`>.

[18] FREE SOFTWARE FOUNDATION, INC. *GCC, the GNU Compiler Collection* [online]. Last revision 2014-04-22, [cit 2014-04-26]. URL: <`http://gcc.gnu.org/`>.

[19] COMEAU COMPUTING. *Comeau C/C++* [online]. Last revision 2013-07-28, [cit 2014-03-31]. URL: <`http://www.comeaucomputing.com/`>.

[20] MCPEAK, Scott. *Elsa: The Elkhound-based C/C++ Parser* [online]. [cit. 2014-03-31]. URL: <`http://scottmcpeak.com/elkhound/sources/elsa/`>.

[21] *Oink: a Collaboration of C/C++ Tools for Static Analysis and Source-to-Source Transformation* [online]. [cit. 2014-03-31]. URL: <`http://daniel-wilkerson.appspot.com/oink/index.html`>.

[22] *PVS-Studio* [online]. Static Code Analyzer for C/C++/C++11. [cit. 2014-03-31]. URL: <`http://www.viva64.com/en/pvs-studio/`>.

[23] *VivaCore library* [online]. [cit. 2014-03-31]. URL: <`http://www.viva64.com/en/vivacore-library/`>.

[24] *The essence of the VivaCore code analysis library* [online]. [cit. 2014-03-31]. URL: <`http://www.viva64.com/en/a/0013/print/`>.

[25] CHIBA, Shigeru. *OpenC++* [online]. [cit. 2014-03-31]. URL: <`http://opencxx.sourceforge.net/`>.

[26] THE CLANG TEAM. *Clang Tidy* [online]. [cit. 2014-04-26]. URL: <`http://clang.llvm.org/extra/clang-tidy.html`>.

[27] QI, Longyi. *OCLint* [online]. [cit. 2014-04-26]. URL: <`http://oclint.org/`>.

[28] *Cppcheck* [online]. A tool for static C/C++ code analysis [cit. 2014-04-26]. URL: <`http://cppcheck.sourceforge.net/`>.

[29] KRZIKALLA, Olaf. *Scout*, A Source-to-Source Transformator for SIMD-Optimizations [online]. last revision 2013-07-24, [cit. 2014-04-26]. URL: <`http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/scout/index_html_en?set_language=en`>.

[30] The Clang Team. *libclang: C Interface to Clang* [online]. [cit. 2014-06-24]. URL: <`http://clang.llvm.org/doxygen/group__CINDEX.html`>.

[31] The Clang Team. *Clang Plugins* [online]. [cit. 2014-04-26]. URL: <`http://clang.llvm.org/docs/ClangPlugins.html`>.

[32] The Clang Team. *LibTooling* [online]. [cit. 2014-04-26]. URL: <`http://clang.llvm.org/docs/LibTooling.html`>.

[33] The Clang Team. *AST Matchers reference* [online]. [cit. 2014-04-26]. URL: <`http://clang.llvm.org/docs/LibASTMatchersReference.html`>.

[34] Kitware. *CMake* [online]. [cit. 2014-04-26]. URL: <`http://www.cmake.org/`>.

[35] Wikipedia. *Profile-guided optimization — Wikipedia, The Free Encyclopedia* [online]. Last revision 2014-04-22, [cit. 2014-04-26]. URL: <`http://en.wikipedia.org/w/index.php?title=Profile-guided_optimization&oldid=605306183`>.

[36] Wikipedia. *Cyclomatic complexity — Wikipedia, The Free Encyclopedia* [online]. Last revision 2014-04-19, [cit. 2014-04-26]. URL: <`http://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=604847346`>.

[37] Wikipedia. *Branch predictor — Wikipedia, The Free Encyclopedia* [online]. Last revision 2014-06-10, [cit. 2014-11-15]. URL: <`http://en.wikipedia.org/w/index.php?title=Branch_predictor&oldid=612349563`>.

[38] *Perforce*, Version Control System [online]. [cit. 2014-04-26]. URL: <`http://www.perforce.com/`>.

List of Figures

List of Tables

Listings

List of Abbreviations

API	application programming interface
AST	abstract syntax tree
CFG	control flow graph
CR	carriage return
CRTP	curiously recurring template pattern
DT	derivation tree
GCC	GNU compiler collection
JSON	JavaScript object nation
LF	line feed
PT	parse tree
PGO	profile guided optimization
RAII	resource acquisition is initialization
SIMD	singe instruction, multiple data

Appendix A

Content of attached media

```
<Optical medium>
 ├─ doc ........................................ Documentation
 │   ├─ thesis ......................... Thesis TEX files and the pdf file.
 │   └─ doxygen ..................... Doxygen generated documentation
 └─ src ........................................ Optimizer source code
```

Appendix B

Output of prefetch method

The diagnostic of the prefetch optimization method outputs information only if there is anything to optimize and the optimizer is not in the *build* mode. Firstly, the method introduce itself and points out to a box definition in source code.

```
[prefetch] optimization of box merge_box
boxes.hpp:55:7: info: declared here:
class merge_box : public bobox::basic_box {
      ^~~~~~~~~
```

Then, for every input that can be optimized by adding prefetch calls, there is a message pointing to the helper macro, the name of the input, and the list of locations where data from this input is likely to be used.

```
boxes.hpp:60:24: info: missing prefetch for input declared here:
BOBOX_BOX_INPUTS_LIST(left,0, right,1);
                      ^~~~
boxes.hpp:99:11: info: used here:
left.eof() && !right.eof()) {
^~~~~~~~~
boxes.hpp:112:11: info: used here:
left.eof()) {
^~~~~~~~~
```

The message with the optimization suggestion looks different for the case when there is `init_impl` overridden in the optimized box or the case when it is not. If there is the overridden initialization function, it points to its location and it suggests adding the call to prefetch member function.

```
boxes.hpp:68:18: suggestion: prefetch input in init:
virtual void init_impl();
             ^~~~~~~~~
```

If there is no initialization function, the optimizer suggests to override it together with prefetch calls.

```
boxes.hpp:55:7: suggestion: override init_impl() in box with
prefetch call(s):
class merge_box : public bobox::basic_box {
      ^~~~~~~~~
```

The output described above is common for *diagnostic* and *interactive* modes. The interactive mode additionally asks a user with the *yes/no* type of a question whether he wants the optimizer to execute the proposed suggestion and transform code. In the *build* mode there are no questions and a code transformation is always applied.

Output of yield complex method

Diagnostic and interactive modes are verbose modes with the same diagnostic output. The problematic part is the diagnostic output itself. Unfortunately, it is very hard to express the reasoning of the optimization algorithm in the text format. The result is that the diagnostic shows only suggestions with information where to put the yield member call expression.

```
[yield complex] optimization of box merge_box

boxes.hpp:70:18: info: method takes too long time on some paths:
virtual void sync_mach_etwas() BOBOX_OVERRIDE
             ^~~~~~~~~~~~~~~
boxes.hpp:87:9: suggestion: placing yield() call just before
statement:
for (int l = 0; l < 100; ++l)
^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

It is up to a programmer to lookup the place in code and investigate why previous code is complex and why it is worth to yield an execution. In the interactive mode, there is also the *yes/no* type of a question whether a user wants to let the optimizer to update code. The chosen answer obviously does not affect the next algorithm work since it only allows a user to select suggestions from the already pre-calculated result. The algorithm has already finished its work.

Appendix C

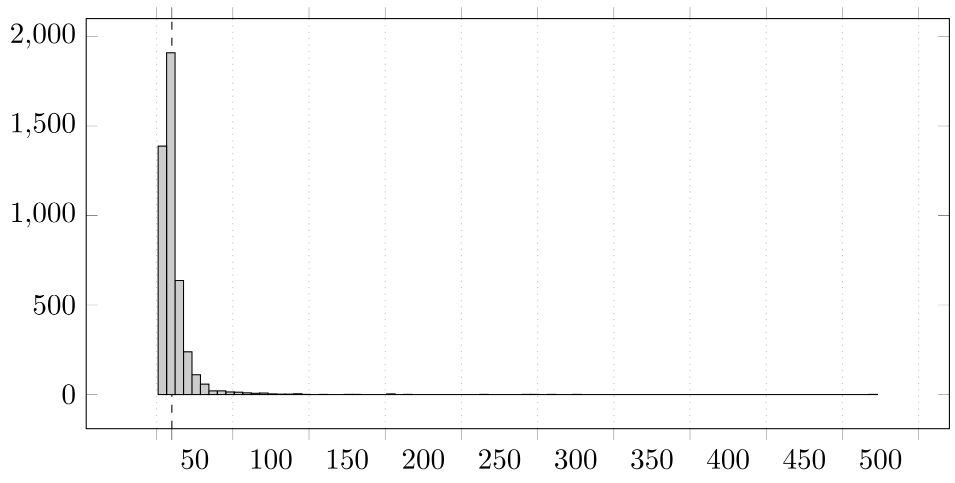
Optimizer loop execution statistics

ID	for			while		
	Executions	Body	Average	Executions	Body	Average
0	87	84	0.97	0	0	0.00
1	28	54	1.93	0	0	0.00
2	25	0	0.00	0	0	0.00
3	13	0	0.00	0	0	0.00
4	13	0	0.00	13	153	11.77
5	13	0	0.00	0	0	0.00
6	0	0	0.00	1	18	18.00
7	0	0	0.00			
8	0	0	0.00			
9	0	0	0.00			
10	0	0	0.00			
11	39	78	2.00			
12	0	0	0.00			
13	39	468	12.00			
14	8	58	7.25			
15	0	0	0.00			
16	0	0	0.00			
17	2178	2033	0.93			
18	21	22	1.05			
19	25	287	11.48			
20	287	690	2.40			
21	690	106	0.15			
22	25	287	11.48			
23	287	690	2.40			
24	680	2178	3.20			
25	491	67	0.14			
26	68	137	2.01			
27	22	242	11.00			
28	22	242	11.00			
29	218	518	2.38			
30	8	8	1.00			
31	8	22	2.75			
32	22	22	1.00			
33	25	92	3.68			
34	25	287	11.48			
35	5	11	2.20			
36	39	468	12.00			
37	468	2340	5.00			
38	18	22	1.22			
39	1	18	18.00			
40	1	1	1.00			
41	0	0	0.00			
42	0	0	0.00			
43	0	0	0.00			
44	0	0	0.00			
45	1	6	6.00			
46	1	1	1.00			
47	1	1	1.00			
Average			**4.19**			**14.88**

Optimizer functions complexities statistics

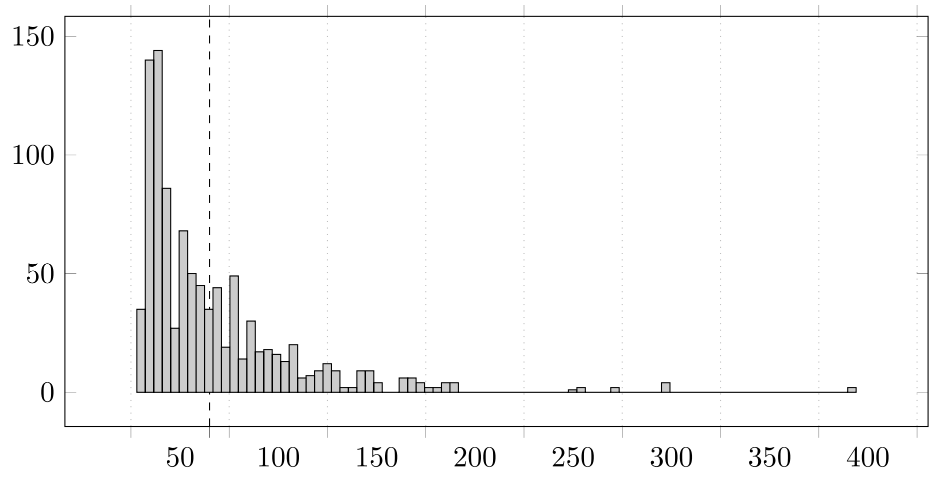
Inlined functions

Functions	4456
Minimal complexity	1
Maximal complexity	473
Average complexity	11.457



Non-inlined functions

Functions	978
Minimal complexity	3
Maximal complexity	369
Average complexity	41.4141



150
100
50
0
50
100
150
200
250
300
350
400