

Challenges and Opportunities in C/C++ Source-To-Source Compilation

João Bispo ✉🏠

University of Porto, Portugal

Nuno Paulino ✉🏠

Faculty of Engineering, University of Porto, Portugal

Luís Miguel Sousa ✉🏠

Faculty of Engineering, University of Porto, Portugal

INESC TEC, Porto, Portugal

Abstract

The C/C++ compilation stack (Intermediate Representations (IRs), compilation passes and backends) is encumbered by a steep learning curve, which we believe can be lowered by complementing it with approaches such as source-to-source compilation. Source-to-source compilation is a technology that is widely used and quite mature in certain programming environments, such as JavaScript, but that faces a low adoption rate in others. In the particular case of C and C++ some of the identified factors include the high complexity of the languages, increased difficulty in building and maintaining C/C++ parsers, or limitations on using source code as an intermediate representation. Additionally, new technologies such as Multi-Level Intermediate Representation (MLIR) have appeared as potential competitors to source-to-source compilers at this level.

In this paper, we present what we have identified as current challenges of source-to-source compilation of C and C++, as well as what we consider to be opportunities and possible directions forward. We also present several examples, implemented on top of the Clava source-to-source compiler, that use some of these ideas and techniques to raise the abstraction level of compiler research on complex compiled languages such as C or C++. The examples include automatic parallelization of `for` loops, high-level synthesis optimisation, hardware/software partitioning with run-time decisions, and automatic insertion of inline assembly for fast prototyping of custom instructions.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Source code generation; Software and its engineering → Development frameworks and environments; Software and its engineering → Software maintenance tools

Keywords and phrases Source-to-source, compilation, transpilers, C/C++, code transformation

Digital Object Identifier 10.4230/OASICS.PARMA-DITAM.2023.2

Category Invited Paper

Funding *Luís Miguel Sousa*: This research has been partially sponsored by the Portuguese Science Foundation (FCT) under research grant SFRH/BD/10002/2022.

Acknowledgements We would like to thank José G. F. Coutinho for reviewing the paper and the useful feedback.

1 Introduction

When writing compiled software in languages such as C and C++, we rely on modern compilation toolchains, such as LLVM and GCC. Such toolchains are incredibly complex pieces of software [15], which are capable of not only correctly translating the code into other languages, usually machine-level, but also of transforming and optimising code, to meet non-functional requirements such as better execution time or smaller code size.



© João Bispo, Nuno Paulino, and Luís Miguel Sousa;
licensed under Creative Commons License CC-BY 4.0

14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023).

Editors: João Bispo, Henri-Pierre Charles, Stefano Cherubin, and Giuseppe Massari; Article No. 2; pp. 2:1–2:15



Open Access Series in Informatics

OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Compiler research at this level is usually done by working directly with the source code of these toolchains, typically by forking existing versions to implement the required modifications. Developers have employed several techniques to improve usability of compilation toolchains, such as well-defined low-level Intermediate Representations (IRs) [26, 40], pluggable compiler passes [47] or Domain Specific Languages (DSLs) that generate code for the toolchain, such as Tablegen [33].

However, there are a number of challenges inherent to this approach. Low-level IRs are extremely important, as a common representation for distinct input languages, but it is common for useful semantic details of the original high-level language to be lost in translation [53]. Traditional compiler IRs are usually tied to a specific computing model (e.g. the von Neumann machine), which can increase the difficulty of using the same IR to target other computation models. Since each compiler has its own IR, custom compiler passes become tied to a specific compiler, and the development flow of modifying a complex tool such as a compiler toolchain makes sharing and reusing custom passes difficult, imposing a significant entry barrier to researchers who are not compiler experts but wish to explore code analyses and transformations.

We consider that this area is ripe for more high-level approaches, and recent developments such as Multi-Level Intermediate Representation (MLIR) [27], part of the Low Level Virtual Machine (LLVM) framework, confirm this vision. In particular, C and C++ source-to-source compilation, as a first step in the compilation toolchain, has been previously proposed as a complementary approach [6, 44, 5, 22], and our experience indicates it can help address these challenges. The ubiquity of C and C++ puts the languages in a special position that justify using them as an IR, in the same sense that compilers traditionally use low-level IRs, but at a higher abstraction level.

However, C and C++ are complex languages, making source-to-source very challenging in this case. We identify several challenges related to C and C++ source-to-source compilation, which include restrictions in the code that can be parsed, difficulties in integration and interaction with traditional compilers, dealing with complex IRs, as well as other competing technologies.

On the other hand, we consider that these problems are not insurmountable, and we also identify possible solutions and opportunities, such as using unmodified established parsers, propose work flows that do not require recompilation or starting from complex codebases, distinguish between human-level and compiler-level use cases (and take advantage of both), and provide high-level environments that promote testing and prototyping.

We have previous experience with source-to-source compilation for C and C++, and we have had the opportunity to implement several of the ideas presented here in our own compiler, Clava. Several works have already used Clava as a way to analyse and transform C and C++ code, and we show several examples of what has been possible after applying these ideas and techniques.

Source-to-source for C and C++ is not new, and many tools have already been developed. Section 2 introduces several of these tools. Section 3 presents the identified challenges, and Section 4 possible solutions and opportunities. Section 5 presents Clava, as well as several works that have used and extended the compiler, and Section 6 concludes the paper.

■ **Table 1** Summary of Source-to-source Compilers for C/C++.

Work	Codebase	Parser	Transformations	Extension mechanism
Clang ¹ [31]	C/C++	Clang	Text-based	Framework
ROSE ² [44]	C/C++	EDG	IR-based	Framework
Insieme ³ [18]	C/C++	Clang	IR-based	Framework
Cetus ⁴ [5]	Java	Custom	IR-based	Framework
Artisan [51]	Python	Clang	Text-based	Interpreter (Python)
CIL ⁵ [37]	C/OCaml	Custom	IR-based	Interpreter (OCaml)
Mercurium ⁶ [6]	C/C++	Custom	IR-based	Framework (dynamically loaded plugins)
Coccinelle ⁷ [28]	C/OCaml	Custom	Text-based	Interpreter (DSL)
Clava ⁸ [9]	Java	Clang	IR-based	Interpreter (JavaScript)

¹<https://github.com/llvm/llvm-project/tree/main/clang> ²<https://github.com/rose-compiler>

³<https://github.com/insieme> ⁴<https://github.com/hkhetawat/Cetus> ⁵<https://github.com/cil-project/cil>

⁶<https://github.com/bsc-pm/mcxx> ⁷<https://gitlab.inria.fr/coccinelle> ⁸<https://github.com/specs-feup/clava>

2 Source-to-Source Compilation

Source-to-source compilers (also commonly called *transpilers*) are tools whose output is code still in a high-level language, and in many cases, the same as the input language. This technology is widely used and quite mature in certain ecosystems, most notably in JavaScript, where it is used, for instance, to provide backwards compatibility of newer language revisions [39].

Source-to-source compilers, after parsing, usually represent the code using an Abstract Syntax Tree (AST) as an intermediate representation. We can generally classify the way C and C++ source-to-source compilers perform transformations in one of two forms, text-based or IR-based. The former uses manipulation of the textual source input, using the AST as a guide; the latter uses manipulation of the AST itself, as an IR, emitting the transformed code directly from the AST. Due to the nature of C and C++, both approaches have uses. In particular, since C and C++ both support a text-based preprocessor, the code the parser receives can be significantly different than the original source-code. This means that to apply transformations that preserve as much as possible the original source code (e.g. IDE refactoring), they should be applied before the preprocessor, directly to the text of the source code. However, this approach prevents the use of the AST as a modifiable IR, and usually requires frequent parsing steps. Using IR-based transformations provide a higher degree of flexibility, as well as a more robust base for building compiler passes over the source code. Such an approach can feed the output directly to the compiler, or feed back information to the source-to-source compiler, which can use it in a text-based transformation.

We can also classify the compilers regarding how a user can implement new transformations. We have identified two categories, frameworks and interpreters. Frameworks allow the implementation of new transformations by using the compiler as a library, and usually writing the transformations in the same language as the language of the codebase. The new transformations are in this way usually bundled inside a new version of the compiler. Interpreters are compilers that besides the source code, also accept as input the transformations to be applied, defined in a language that can be different from the language of the codebase of the compiler. In this approach, the compiler does not need to be modified to execute new transformations.

2.1 Source-to-Source Compilers for C and C++

Table 1 summarizes the characteristics of several notable source-to-source compilers for C/C++. Nearly all are openly accessible, and based on AST manipulation. But often they are limited to a subset of C/C++, require modifying and recompiling internal codebases, and/or are designed with a specific set of transformations in mind.

- Clang [31] itself provides some text-based source-to-source capabilities. Specifically, Clang’s *libTooling* library provides a `Rewriter` class that can be used to manipulate the source files[43]. User specified transformations are written in C/C++, and invoke *libTooling* as an API that uses the AST generated by the Clang parser to navigate the code. By matching AST patterns, approaches such as auto-vectorization [25] or insertion of OpenMP boilerplate from templates [7] can be achieved. Access to the Clang AST provides precise manipulation capabilities, but requires expertise on compiler concepts, and is therefore geared in particular towards developers already familiar with the Clang/LLVM ecosystem. Finally, in contrast to most approaches in Table 1, the source code is modified by re-writing the input file directly, rather than re-emitting code from a modified AST. This preserves any pre-processor macros present in the input code;
- The ROSE [44] compiler supports C/C++ and FORTRAN (and others), and supports generic AST-based transformations over its own IR, generated by a parser based on Edison Design Group (EDG)’s front-end[1]. It is itself implemented in C/C++, and supported by an additional tool, ROSETTA, to (re-)generate the IR if needed. User transformations are implemented by direct manipulation of the ROSE IR using the provided C++ APIs. Examples of transformations already present in the compiler are auto-parallelization of loops, as well as optimizations such as loop fissioning and fusion;
- The Insieme infrastructure [18, 22] first parses the input C/C++ into its own IR, INSPIRE [23], which is generated from the AST produced by the Clang parser. INSPIRE is designed to expose parallelism explicitly. The backend generates transformed C/C++ (optionally OpenCL) which interacts with the Insieme runtime, used to dispatch workloads onto parallel resources. Thus Insieme is specifically geared to transform sequential code onto parallel oriented paradigms, specifically, thread oriented workloads;
- Cetus [5] is written in Java and also supports AST-based transformations. Its primary purpose is automatic optimization of a supported subset of ANSI C, specifically for automatic parallelization. Cetus internally implements a set of ten transformations for this effect (five general optimization passes, and five parallelization passes), which have shown to produce improvements when applied to the NAS Parallel Benchmarks [48], versus manually parallelized versions. These transformations are part of the Cetus Java codebase, and to implement new custom transformations one needs to fork and extend the compiler;
- Artisan [51] is a Python3 package focused on providing source-to-source compilation for heterogeneous platforms. It uses Clang to parse the code, and accepts analysis and transformations implemented as Python scripts. Its main use case is hardware/software partitioning targeting CPU + Field-Programmable-Gate-Array (FPGA) systems. Namely, Artisan aims to automate the application of known design patterns and optimizations that are required for performance maximization when targeting parallel oriented computing paradigms. Some integration issues are addressed, by abstracting High-Level-Synthesis (HLS) tools, their invocations, and resulting artifacts as Python objects. Notably, Artisan can generate OpenCL work-group oriented code from agnostic C/C++;

- The C Intermediate Language (CIL) [37] approach recognizes that C/C++ contains many complex constructs, hampering a straightforward analysis of source code. The aim of CIL is to convert C code to a representation that, while not being a proper subset, is close to C, and easier to work with. It uses a custom parser that supports ANSI C, including custom Microsoft and GNU extensions, and generates a high-level representation that preserves most semantic information of the code. The representation contains simplifications such as removing redundant constructs and syntactic sugar, making implicit casts explicit, and separating value evaluation, side-effect creation, and control-flow changes. It also incorporates a Control Flow Graph into the representation to simplify the analysis. After this conversion, it applies any transformations that the user has specified, using an embedded DSL in OCaml [30], and outputs the transformed program;
- Mercurium [6] is a source-to-source infrastructure developed by the Barcelona Supercomputing Center, based on a custom parser that supports C/C++ and Fortran, and uses a common shared IR. It is one component of a framework for OpenMP based parallelisation, capable of retargeting code to GPUs (i.e, CUDA) as well as FPGAs [11]. Mercurium is designed as a platform for fast testing and development of new OpenMP extensions, but it is extensible and has been used to implement other computing models. New transformations are given to Mercurium as plugins written in C/C++, and loaded at runtime to act as compiler passes;
- Coccinelle [29, 28] was created in 2006 in the specific context of maintaining the Linux kernel, and has since been extensively used. It is based on a DSL whose syntax is inspired by diff logs, and can express semantic patches to be applied throughout the entire codebase. The transformations are text-based, as it uses pattern matching rules to replace, for instance, certain API call changes that occur due to implementation changes in underlying device drivers. Multiple pattern matching rules can be applied in sequence, on one input C file at a time. Although designed for a very specific use-case, its strong adoption and impact on the maintenance of the Linux kernel illustrates the potential of source-to-source tooling;
- The Clava compiler [9] is built on top of the LARA framework [41], which enables the specification of complex code analyses and transformations via JavaScript scripts. Like Cetus, it is implemented in Java. It relies on an unmodified version of Clang's parser to generate a C/C++ AST that is very similar to Clang's [31], but extended to allow for transformations to be applied directly to the AST. New transformation passes can be specified as separate JavaScript files processed by Clava, without the need of modifying Clava itself;

Despite these efforts, a number of challenges persist. We detail them in the following section.

3 Challenges

Source-to-source compilation of C and C++ presents several challenges, which some authors have previously identified. For instance, Milewicz et al. [35] focus on the limitations that source-to-source tools present in an HPC environment. Although the work is not specifically about C and C++, these languages are also widely used in HPC, so several of the presented challenges apply. Next are the main shortcomings of C and C++ source-to-source compilation that we have identified, based on several of the tools and works in the state of the art.

3.1 Limited support for the input languages

It is common for many source-to-source tools to implement their own parsers, in order to have greater control over the generated IR, which usually is an AST. However, C, and in particular C++, are very complex languages, which are still in active development [20, 21]. Often, many C and C++ source compilers support only a limited subset of the language or a specific standard (e.g. a commonly supported standard is ANSI C [5]),

Use of C-style macros and C++ templates also increases parsing difficulty, since they can be complex and not fully supported by custom parsers, or not obvious how to handle in a source-to-source context. In an evaluation of OpenMP performance measurement mechanisms, Huck et al. remark on the difficulty a source-to-source based mechanism had when dealing with C macros [19].

3.2 Integration with existing toolchains

Since code transformations must be applied before compilation, it is not clear how to efficiently integrate a source-to-source step into standard toolchain. Additionally, since the source-to-source compiler is a separate tool from the compilation framework, the C and C++ compiler will most likely not be aware of the source-to-source transformations.

When implementing a high-performance library for statistical phylogenetics, Ayres et al. opted to implement a C API integrated into the language, rather depending on an external tool that needs to translate the code [4]. Alternatively, McCormick et al. propose a DSL, as an extension of C and C++, that allows to define and operate over mesh data types [34], and that is integrated along the several stages of the LLVM framework, from the parser (Clang) to the debugger (LLDB). They mention that their solution has several advantages over source-to-source approaches, such as keeping domain-specific information along the toolchain and better support for debugging, although they recognise their approach is more complex than an equivalent source-to-source one.

3.3 Unintended interactions with the compiler

Since source-to-source analyses and transformations are applied before compilation to lower abstraction levels, it might be unclear how source transformations will affect compiler-driven optimization passes in a general case.

For instance, Denis et al. [12] measures numerical accuracy by replacing standard floating point operations with equivalent ones that use Monte Carlo Arithmetic. They observe that source-to-source approaches are not able to capture the influence of compiler optimizations on the numerical accuracy, since replacing the standard operations with library calls prevents such optimizations.

Although not exclusive to source-to-source approaches, Kruse et al. [24] point out that polyhedral loop optimisations, while very promising, usually are not activated by default in standard optimisation levels (e.g. -O3) because they are applied before other compiler passes and interfere with them. In particular, they refer to the introduction of scalar dependencies by the polyhedral optimisations that the Single Static Assignment (SSA) representation does not handle well. Additionally, since in this case the polyhedral optimizations are done at the beginning of the pipeline, no passes such as inlining have been applied yet, which limits the applicability of the optimisations to small loops.

3.4 Limitations in source code as an IR

Low-level IRs strive for a level of parsimony that allows to reduce complexity when handling and transforming them. In this regard, languages such as C and C++ are in comparison more complex, with a larger number of constructs. This increases the difficulty of using them as an IR, since there are more cases to consider when creating analyses and transformations, which also reduces generality.

Besnard et al. [8] propose a library that adds support for a shared-memory paradigm via threads in an MPI context, which is a distributed-memory paradigm, in order to use an MPI-only solution for both local and distributed communication, instead of a mixed solution (e.g. MPI + OpenMP). One of the necessary modifications is to privatise shared, global variables, and although they say that a source-to-source approach would improve the portability of the solution, they refer that it requires elaborate data-flow analyses done over complex data-types and potential indirect references.

Similarly, Adamski et al. [2], which proposes an heuristic for polyhedral analysis with run-time information, mentions they chose to implement their approach in LLVM-IR instead of at the source-level due to features such as SSA representation and more explicit data dependencies and control flow.

3.5 Competing technologies

A recent contribution to the compiler research space is MLIR [27], as part of the LLVM project. This novel approach introduces an intermediate representation that aims at solving certain shortcomings of LLVM-IR related to targeting non-conventional computing models and heterogeneous architectures. MLIR provides an SSA based, recursively-nested IR whose semantics are encoded in user-defined *dialects*, which encapsulate operations, data type schemata and transformations within the same and between other dialects. This technology allows the reuse of many kinds of compiler passes, across several abstraction levels. There is one preferential direction in the transformations (i.e., lowering transformations), but recent works address the opposite flow, i.e., raising transformations [36, 10].

The entry point has mainly been high-level DSLs that can be lowered to several targets (e.g., LLVM-IR, CUDA, HDL), but there is an increased interest in providing MLIR parsers and dialects for languages such as C/C++ [32]. Together with MLIR's ability of moving between abstraction levels, it can be considered as a potential competing technology to source-to-source compilers.

4 Opportunities

We consider that several of the challenges presented in Section 3 are not insurmountable, and that there are opportunities for better and more accessible source-to-source compilers for C and C++, which will allow novel workflows and applications.

4.1 Reuse of existing parsers as-is

As mentioned in Section 3.1, limited support of the C and C++ languages is a common issue. Most of this limitation stems from tools using custom parsers [5][14]. When developing a C or C++ source-to-source compiler, we consider that in almost all cases, parsing the language should be offloaded to third-party libraries, and the use of custom parsers should be avoided. Parsing C and C++ is a very difficult problem that should be handled by projects dedicated to this task.

Several tools already do this. In particular ROSE [44], arguably one of the most successful C and C++ source-to-source compilers, since the beginning has used the EDG's proprietary C++ front-end as a parser, while more recent approaches tend to use Clang as a front-end [18, 51]. Additionally, we think it is highly recommended that the third-party parsers are used as-is, with no modifications. Since C and C++ are still evolving languages, this allows an easier update path, when new standards or language features appear.

4.2 Improved composability and compatibility

Since the external interface of source-to-source compilers is the target language itself, such tools should take advantage of this and provide easy and seamless integration with compilation environments and toolchains. Compiler toolchains for compiled languages such as C and C++ are already a collection of many different tools that are called back-to-back. Source-to-source compilers can be easily integrated in such a flow, as another tool in the toolchain (e.g. Insieme provides a driver that works as a drop-in replacement for calls to the GCC or Clang driver [18]).

We also advocate for approaches that extend the compiler without the need to change the compiler itself, e.g. through APIs that the compiler interprets, or plugins that can be dynamically loaded. A user should be able to download a given custom library that is immediately ready for use, similar to how we are able to seamlessly use third-party APIs in most modern programming languages (e.g. Maven dependencies in Java, Pip in Python, npm in JavaScript).

We consider such an approach can provide better support for composing different works from different authors, when compared with an approach that requires modification of the compiler toolchain itself and distribution of a custom executable. It allows users to simply pick and choose from existing solutions, and ideally, use the same system to easily implement and integrate their own analyses and transformations.

This composability can be extended to the use of different source-to-source compilers. Tools that target the same language are most likely compatible with each other by default, as long as they support the language constructs present in the source code, which allows further possibilities in mix and match scenarios.

4.3 Widening the scope and taming complexity

Usually source-to-source compilers are used in what we can call human-level use cases, that is, automating transformations a human programmer would do if they had the resources or experience to do themselves directly over the source code (e.g., recursive functions to iterative models, array flattening, loop interchange). Usually the output is code that is still readable by humans. We consider that source-to-source compilers should also embrace what we can call compiler-level use cases, where the source-code is treated as a low-level IR, where several compiler passes are applied, changing the code as much as needed. The resulting code is not necessarily seen by a human, and can go directly to the compiler. The two approaches are not mutually exclusive. A user can apply compiler-level techniques that dramatically change the code, in order to extract information, and then discard the changes and use the extracted information in human-level techniques (see Section 5.1).

To use C or C++ as an IR where compilation passes can be applied, it is important to deal with the complexity of the languages. One way to handle this is for source-to-source compilers to provide normalisation or canonicalization passes, similar to what traditional compilers do for lower-level IRs. Such passes can be very generic and easily reused, and can significantly reduce the complexity of using C or C++ as an IR (see Section 5.4).

One of the advantages that is often pointed out about using mature compilation frameworks such as GCC or LLVM is the possibility to reuse several analyses and transformations that are already implemented. The same principle can be applied to source-to-source frameworks, if they allow simple reuse of compilation passes, in particular if the observations in Section 4.2 are followed. Several of the same transformations that are usually done by a compiler in low-level IRs can be useful if available on a source-to-source level (see Section 5.1, which extensively uses inlining during analysis to increase the number of loops that can be parallelized).

Other ways to tame complexity in source-to-source approaches include minimizing the quantity of automatically inserted code by using instead libraries and inserting code to call them, or providing high-level abstractions that hide the complexities of the language (e.g., APIs for inserting instrumentation code [42]).

4.4 Testing and Prototyping Environments

We consider it is crucial to have a testing environment that allows to easily and quickly test source-to-source transformations. Since source-to-source tools usually are the first step in a compilation toolchain, they are in a privileged position in such flows, opening the possibility for integrated environments where parsing, analysing, transforming, compiling and executing an application can be accomplished from within the same script. Such environments naturally allow design-space exploration (DSE) loops, and the possibility of exploring strategies that use run-time information, at any level of the compiler toolchain [38].

This can also be the base for a prototyping environment for compiler transformations that is lighter than going directly to a traditional framework. An initial implementation can start as a source-to-source transformation, for testing and validation (see Section 5.4), and after the work reaches a certain level of maturity, it is developed and integrated in a traditional compiler toolchain. Also, the same environment can be used to implement very specialised transformations that might not justify integration into a traditional compiler framework, and that can be easily enabled or disabled according to the target compiler or machine.

4.5 Make compilers in general more accessible

Some of the opportunities presented here are not limited to source-to-source compilers, but could potentially be applied to low-level compilers in general. Take for instance, the MLIR technology, which is a C++ framework that should be used as a library to build your own compiler. This is expected since, similar to LLVM, it is a framework for building compilers. However, taking into account how extensible MLIR is, it is in a privileged position to provide mechanisms such as the ones mentioned in Section 4.2.

Currently, there are three main methods to use or extend MLIR: C++, Operation Definition Specification (ODS) and Python. Since MLIR is a C++ framework, we can directly write heavily templated C++ code to implement our own dialects, which mainly contain operations and transformations, but can also contain custom types. This can be quite cumbersome, so MLIR supports defining operations and data types using a DSL, TableGen¹, that generates MLIR-compatible C++ code. Finally, there are Python bindings that allow inspecting and transforming the IR with existing dialects, but not defining new dialects.

Although MLIR is a noticeable improvement in accessibility regarding LLVM (and LLVM itself also improved upon its predecessors, such as GCC), we consider there is still considerable room for improvement, for instance, by providing environments such as the ones proposed in

¹ <https://llvm.org/docs/TableGen/>

Section 4.4. There are already works that tackle these issues, such as Vasilache et al.[52] which, among other things, propose an embedded DSL for Python that allows the creation of MLIR operations from within Python.

Finally, although the technology is not there yet, it can become a very interesting framework for creating source-to-source compilers. This can also provide a means of going beyond the LLVM ecosystem, for use cases where the target compiler is not under the developer’s control (e.g. embedded systems).

5 Illustrating C/C++ Source-to-Source with Clava

The previously mentioned Clava compiler is our own work on source-to-source for C/C++ (as well as other C-like languages, i.e., OpenCL, CUDA) [9]. As we have used it to address some of the challenges and opportunities outlined previously, we now provide some additional details as well as example use-cases.

Clava relies on an unmodified version of Clang’s parser to generate its own IR, the Clava AST. This IR is very similar to Clang’s AST, albeit with some differences. Besides some normalization steps (e.g. nodes such as `if`, `for`, *etc* always contain a scope block as a child), the main difference is that the Clava AST is built to be modified and emit the equivalent C/C++ code that its current structure represents. The decision to use Clang as-is proved to be fruitful, Clava has gone through two Clang updates (from 3.8 to 7, and from 7 to 12) with a reduced number of modifications.

To extend Clava, one does not need to change the compiler (i.e., modify Clava’s own codebase). Instead, custom analyses and transformations are written as JavaScript scripts, which Clava interprets and applies over a given source-code. The scripts represent a standard JavaScript programming environment that has access to the Clava AST, as well as having access to source-to-source specific APIs, such as instrumentation, or compiling and executing the modified code from within the script. New APIs can be added by specifying, as a configuration parameter, new include folders to other JavaScript files. Additionally, Clava is a cross-platform Java application that does not require installation or dependencies, and provides a CMake² package which applies the scripts to any C/C++ CMake project with very little effort.

A brief example of these capabilities is shown in Listing 1. The JavaScript APIs provided by Clava allow for selection, analysis and modification of C/C++ code constructs, such as functions, loops, *if-elses*, *etc*. For this example, the transformation specifies that the input code, shown in Listing 2, should be queried for a function named *foo*, and then return a list of all loops within the function body. For each loop, a comment is inserted prior to the loop in the output code, shown in Listing 3. The comment includes the line number of the loop in the original code. The function `insertBefore()` also accepts strings, in case we want to insert literal code, however, we consider that it is preferable to create and insert nodes.

```
laraImport("weaver.Query")
laraImport("clava.ClavaNodes");

for(const loop of Query.search("function", {name: "foo"}).search("loop")) {
  const commentNode = ClavaNodes.comment(" Loop at line " + loop.line)
  loop.insertBefore(commentNode)
}
```

■ Listing 1 Javascript file defining a Clava transformation to insert comments prior to loops.

² João Bispo, 2021, “Clava CMake Package”, Github Repository, <https://github.com/specs-feup/clava/tree/master/CMake>

```

1  int foo() {
2      int a = 0;
3
4      for(int i = 0; i < 100; i++)
5          a += i * i;
6
7      for(int i = 0; i < 100; i++)
8          a += i + 1;
9      return a;
10 }
```

■ **Listing 2** Example prior to comment insertion.

```

int foo() {
    int a = 0;
    // Loop at line 4
    for(int i = 0; i < 100; i++)
        a += i * i;
    // Loop at line 7
    for(int i = 0; i < 100; i++)
        a += i + 1;
    return a;
}
```

■ **Listing 3** Example after comment insertion.

We chose to rely on externally specified transformations in a widely adopted language such as JavaScript in order to lower the entry barrier and raising the abstraction level for compiler research. Next we present several examples that have used Clava to automatically analyse and transform code. Most of these works have extended Clava with new APIs implemented in JavaScript and, excluding the first example, they have been developed in the context of MSc theses.

5.1 AutoPar – Automatic Parallelisation of for Loops

AutoPar [3] is a Clava library that statically detects if a `for` loop can be parallelized or not, and if it determines that it can, generates an OpenMP pragma for the loop. This is an example that mixes human-level and compiler-level approaches. Initially, the code is heavily transformed, by inlining as many calls as possible in all loop bodies. The transformed code is then analysed and tested for parallelism. All changes are then discarded, and the collected information is used to generate OpenMP pragmas, which are inserted in the original code.

5.2 Insertion of High-Level-Synthesis Directives

Recent High-Level-Synthesis (HLS) tools such as Xilinx’s Vitis HLS generate hardware implementations of C/C++ functions, circumventing traditional hardware design via Verilog or VHDL. However, some expert knowledge is still required, as the HLS compiler cannot (currently) fully infer design intent or identify parallelism opportunities from what is, intrinsically, sequentially oriented code. Santos et al. [45, 46] use Clava to automatically insert *pragmas* which Xilinx’s HLS compiler uses as optimisation hints to generate better performing hardware implementations, avoiding the need for expert know-how, as well as design effort.

5.3 Tribble – Targeting Heterogeneous Systems

Tribble [49] is a Clava library for retargeting C/C++ applications to FPGA based heterogeneous systems. Target functions are identified with a single *pragma* statement, which Tribble first optimises and then passes to Xilinx’s High-Level-Synthesis. The original code is modified with the required OpenCL API boilerplate to invoke the generated circuit, while retaining the original software version. A user-defined scheduler [50] is also inserted to select, at runtime, which version (CPU or FPGA) to call based on, e.g., estimated compute workload.

5.4 Inline Assembly Insertion – RISC-V Custom Extensions

Henriques [16], developed a Clava API capable of rewriting C `for` loops marked with a user-specified *pragma* [17] as inline assembly that uses UVE [13] instructions, a custom RISC-V instruction extension for streaming and vectorization. To do this, the Clava AST was used

as a traditional compiler IR, where a series of normalization steps were applied. The code was transformed to a functionally equivalent SSA-like format, facilitating the identification of streaming and vectorization patterns which map to UVE instructions. A final step inserts the inline assembly code, allowing for automatic generation of UVE assembly code from C code without forking an existing compiler. We consider that such an approach can contribute to faster prototyping of new extensions, and reduce manual assembly programming effort during the early stages of development.

6 Conclusions

In this paper we have provided a distillation of the insights acquired during several years working in C and C++ source-to-source compilation. We presented a short summary on several notable source-to-source compilers for C and C++, highlighted challenges relative to its further development and adoption, and opportunities that show potential for further work in this area. We also present how several of these ideas have been implemented in our own C/C++ source-to-source compiler, Clava.

We argue that conventional compilation approaches still have significant entry barriers and that source-to-source compilation can have a complementary role in bringing new people to the area. Furthermore, we consider that several of the techniques used to lower the entry barrier of source-to-source could also be applied to traditional compiler development. We also see potential for source-to-source compilation to be applied in scenarios that have been mostly exclusive to low-level IRs.

References

- 1 J Stephen Adamczyk and John H Spicer. Template instantiation in the EDG C++ front end. *Edison Design Group Technical Report*, 1995.
- 2 Dominik Adamski, Michał Szydłowski, G Jabłoński, and J Lasoń. Dynamic tiling optimization for polly compiler. *International Journal of Microelectronics and Computer Science*, 8(4), 2017.
- 3 Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *J. Supercomput.*, 76(9):6753–6785, September 2020. doi:10.1007/s11227-019-03109-9.
- 4 Daniel L Ayres and Michael P Cummings. Heterogeneous hardware support in beagle, a high-performance computing library for statistical phylogenetics. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, pages 23–32. IEEE, 2017.
- 5 Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, Samuel P Midkiff, H Bae, D Mustafa, J w Lee, H Lin, R Eigenmann, S P Midkiff, and C Dave. The cetus source-to-source compiler infrastructure: Overview and evaluation. *Int J Parallel Prog*, 41:753–767, 2013. doi:10.1007/s10766-012-0211-z.
- 6 Jairo Balart, Alejandro Duran, Marc González, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, volume 8, page 2004, 2004.
- 7 G.D. Balogh, G.R. Mudalige, I.Z. Reguly, S.F. Antao, and C. Bertolli. Op2-clang: A source-to-source translator using clang/llvm libtooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 59–70, 2018. doi:10.1109/LLVM-HPC.2018.8639205.
- 8 Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D Maloney. Introducing task-containers as an alternative to runtime-stacking. In *Proceedings of the 23rd European MPI Users’ Group Meeting*, pages 51–63, 2016.

- 9 João Bispo and João M.P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, July 2020. doi:10.1016/j.softx.2020.100565.
- 10 Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, and Henk Corporaal. Progressive raising in multi-level ir. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 15–26. IEEE, 2021.
- 11 Juan Miguel de Haro, Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Ompss@fpga framework for high performance fpga computing. *IEEE Transactions on Computers*, 70(12):2029–2042, 2021. doi:10.1109/TC.2021.3086106.
- 12 Christophe Denis, Pablo De Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. *arXiv preprint*, 2015. arXiv:1509.01347.
- 13 Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. Unlimited vector extension with data streaming support. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 209–222, 2021. doi:10.1109/ISCA52012.2021.00025.
- 14 Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. Mercurium: Design decisions for a s2s compiler. In *Cetus Users and Compiler Infrastructure Workshop in conjunction with PACT*, volume 2011, 2011.
- 15 Dick Grune, Kees Van Reeuwijk, Henri E Bal, Criel JH Jacobs, and Koen Langendoen. *Modern compiler design*. Springer Science & Business Media, 2012.
- 16 Luís Miguel Henriques. Automatic streaming for risc-v via source-to-source compilation. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2022. URL: <https://hdl.handle.net/10216/142750>.
- 17 Miguel Henriques. Clava based transforms for uve code insertion, 2022. URL: <https://github.com/MiguelPedrosa/Dissertacao>.
- 18 Bernhard Höckner. The insieme compiler frontend: A clang-based C/C++ frontend. Msc thesis, University of Innsbruck, 2014. URL: <https://typeset.io/pdf/the-insieme-compiler-frontend-a-clang-based-c-c-frontend-a1djb93945.pdf>.
- 19 Kevin A Huck, Allen D Malony, Sameer Shende, and Doug W Jacobsen. Integrated measurement for cross-platform openmp performance analysis. In *International Workshop on OpenMP*, pages 146–160. Springer, 2014.
- 20 ISO. *ISO/IEC 9899:2018 Information technology – Programming languages – C*. International Organization for Standardization, Geneva, Switzerland, June 2018.
- 21 ISO. *ISO/IEC 14882:2020 Information technology – Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, December 2020.
- 22 Herbert Jordan. *Insieme: A Compiler Infrastructure for Parallel Programs*. PhD thesis, University of Innsbruck, August 2014. URL: <https://diglib.uibk.ac.at/ulbtirolhs/download/pdf/179200>.
- 23 Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 7–18. IEEE Press, 2013.
- 24 Michael Kruse and Tobias Grosser. Delicm: scalar dependence removal at zero memory cost. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 241–253, 2018.
- 25 Olaf Krzikalla. Performing Source-to-Source Transformations with Clang, 2013. European LLVM Conference, Paris. URL: <https://llvm.org/devmtg/2013-04/krzikalla-slides.pdf>.
- 26 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

- 27 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. *CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–14, February 2021. doi:10.1109/CGO51591.2021.9370308.
- 28 Julia Lawall. Coccinelle: Reducing the barriers to modularization in a large c code base. In *Proceedings of the companion publication of the 13th international conference on Modularity, MODULARITY '14*, pages 5–6, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2584469.2584661.
- 29 Julia Lawall and Gilles Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18*, pages 601–613, USA, 2018. USENIX Association.
- 30 Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml System Release 4.14, 2022. URL: <https://v2.ocaml.org/manual/>.
- 31 LLVM Project. Clang: a C language family frontend for LLVM, 2022. URL: <https://clang.llvm.org/>.
- 32 Bernardo Cardoso Lopes and Nathan Lanza. [RFC] An MLIR based Clang IR (CIR) – Clang Frontend – LLVM Discussion Forums. 2022. Available at <https://discourse.llvm.org/t/rfc-an-mlir-based-clang-ir-cir/63319>, 2022. Accessed 2022-07-05.
- 33 Bruno Cardoso Lopes. Understanding and writing an llvm compiler back-end. In *ELC'09: Embedded Linux Conference, 2009*, 2009.
- 34 Patrick McCormick, Christine Sweeney, Nick Moss, Dean Prichard, Samuel K Gutierrez, Kei Davis, and Jamaludin Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *2014 fourth international workshop on domain-specific languages and high-level frameworks for high performance computing*, pages 1–10. IEEE, 2014.
- 35 Reed Milewicz, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review. In *International Conference on High Performance Computing*, pages 233–246. Springer, 2021.
- 36 William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59. IEEE, 2021.
- 37 George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2304:213–228, 2002. URL: https://link.springer.com/chapter/10.1007/3-540-45937-5_16.
- 38 Ricardo Nobre, João Bispo, Tiago Carvalho, and João MP Cardoso. Nonio—modular automatic compiler phase selection and ordering specialization framework for modern compilers. *SoftwareX*, 10:100238, 2019.
- 39 Chris Northwood. Javascript. In *The Full Stack Developer*, pages 159–208. Springer, 2018.
- 40 Diego Novillo. GCC an architectural overview, current status, and future directions. In *Proceedings of the Linux Symposium*, volume 2, page 185, 2006.
- 41 Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. Lara as a language-independent aspect-oriented programming approach. In *Proceedings of the Symposium on Applied Computing*, pages 1623–1630, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3019612.3019749.
- 42 Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João MP Cardoso. Aspect composition for multiple target languages using lara. *Computer Languages, Systems & Structures*, 53:1–26, 2018.
- 43 LLVM Project. Using Clang as a Library – LibTooling, 2022. URL: <https://clang.llvm.org/docs/LibTooling.html>.

- 44 Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.
- 45 Tiago Santos. Acceleration of applications with fpga-based computing machines: Code restructuring. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2020. URL: <https://hdl.handle.net/10216/128984>.
- 46 Tiago Santos and João M.P. Cardoso. Automatic selection and insertion of hls directives via a source-to-source compiler. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 227–232, 2020. doi:10.1109/ICFPT51103.2020.00039.
- 47 Suyog Sarda and Mayur Pandey. *LLVM essentials*. Packt Publishing Ltd, 2015.
- 48 S. Satoh. NAS Parallel Benchmarks 2.3 OpenMP C Version, 2000. URL: <http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp>.
- 49 Luís Sousa. Runtime management of heterogeneous compute resources in embedded systems. Msc thesis, Faculdade de Engenharia, Universidade do Porto, Porto, Portugal, 2021. URL: <https://hdl.handle.net/10216/137152>.
- 50 Luís Miguel Sousa, Nuno Paulino, João Canas Ferreira, and João Bispo. A flexible hls hoeffding tree implementation for runtime learning on fpga. In *2022 IEEE 21st Mediterranean Electrotechnical Conference (MELECON)*, pages 972–977, 2022. doi:10.1109/MELECON53508.2022.9843092.
- 51 Jessica Vandebon, Jose GF Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. Artisan: A meta-programming approach for codifying optimisation strategies. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 177–185. IEEE, 2020.
- 52 Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction. *arXiv preprint*, 2022. arXiv:2202.03293.
- 53 Peter Zangerl, Herbert Jordan, Peter Thoman, Philipp Gschwandtner, and Thomas Fahringer. Exploring the semantic gap in compiling embedded dsls. *ACM International Conference Proceeding Series*, pages 195–201, July 2018. doi:10.1145/3229631.3239371.