SoftwareX 12 (2020) 100565

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

Clava: C/C++ source-to-source compilation using LARA

João Bispo *, João M.P. Cardoso

University of Porto, Faculty of Engineering / INESC-TEC, R. Dr. Roberto Frias s/n, 4200-465 Porto, Portugal

ARTICLE INFO

Article history:
Received 14 June 2019
Received in revised form 10 May 2020
Accepted 3 July 2020

Keywords:
Source-to-source
C/C++
LARA
Compilers

ABSTRACT

This article presents Clava, a Clang-based source-to-source compiler, that accepts scripts written in LARA, a JavaScript-based DSL with special constructs for code queries, analysis and transformations. Clava improves Clang's source-to-source capabilities by providing a more convenient and flexible way to analyze, transform and generate C/C++ code, and provides support for building strategies that capture run-time behavior. We present the Clava framework, its main capabilities, and how it can been used. Furthermore, we show that Clava is sufficiently robust to analyze, instrument and test a set of large C/C++ application codes, such as GCC.

Code metadata

Current code version	v4.2.7
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_209
Legal Code License	Apache-2.0
Code versioning system used	git
Software code languages, tools, and services used	Java, C++, Clang/LLVM, LARA
Compilation requirements, operating environments & dependencies	Java 11
If available Link to developer documentation/manual	http://specs.fe.up.pt/tools/clava/doc/
Support email for questions	jbispo@fe.up.pt

Software metadata

Current software version	v4.2.7
Permanent link to executables of this version	http://specs.fe.up.pt/tools/clava_v4.2.7.zip
Legal Software License	Apache-2.0
Computing platforms/Operating Systems	x86: Ubuntu, CentOS/Fedora, MacOS, Windows; ARM7: Ubuntu
Installation requirements & dependencies	Java 11
If available, link to user manual – if formally published include a reference to the publication in the reference list	https://web.fe.up.pt/~specs/projects/antarex/book/DSL_and_Source_to_Source_Compilation__the_Clava_LARA_Approach.pdf
Support email for questions	jbispo@fe.up.pt

1. Motivation and significance

Trade-offs are, arguably, one of the cornerstones of engineering. When writing software there are many trade-offs to consider. For instance, code can be written to be modular or efficient, to be generic or specialized to a given platform, to save memory or energy consumption. There is no "unique" correct software implementation, and each implementation is instead highly contingent on the non-functional requirements [1] and target platforms. In addition, changes related to performance and efficiency

* Corresponding author.
E-mail address: jbispo@fe.up.pt (J. Bispo).

https://doi.org/10.1016/j.softx.2020.100565

usually require a profiling step, as well as several test and what-if-analysis steps. This translates to a development loop where code is modified, compiled and executed, which usually is mostly manual, error-prone, and time consuming.

Source-to-source compilers can be a valuable tool for generating and managing several code versions, and can enable features such as instrumentation, code transformations (e.g., loop transformations), and code refactoring. In particular, the C/C++ languages lack features (e.g., comprehensive reflection support) that a source-to-source compiler can complement. For a source-to-source compiler to successfully deal with custom tasks it is important that it provides an easy-to-use programmable interface, that allows a developer to implement its strategies. Directly programming a compiler (e.g., Clang, GCC) with custom transformations and analysis passes can require a significant amount of effort, and recent approaches have proposed the use of Domain Specific Languages (DSLs) coupled to a source-to-source compiler. An advantage of DSLs is that they have the potential to express a solution to the domain problem more concisely and in a clearer way. This can also diminish errors or inefficient idioms, and make code generation more efficient. On the downside, DSLs introduce the need of having to learn a new language, and usually are not as integrated in the compilation tool-flow as native libraries or compiler-supported extensions. Despite that, DSLs have been shown to guide compilation flows [2,3], and to describe compiler strategies for guiding source-to-source compilers.

There is a myriad of source-to-source compilers with distinguishing features, mainly regarding the supported code transformations and the accepted input programming languages. For instance, Cetus [4] supports ANSI C, ROSE [5] supports C, C++, Fortran, OpenMP, and UPC, PIPS [6] supports C and Fortran, and Insieme [7] supports C, with plans to support C++. Regarding the capability of exposing to users and developers mechanisms to apply specific strategies and/or to define their own strategies, the approaches have typically focused on specific goals. For example, CHiLL [8] is a DSL to specify sequences of predefined transformations and constrains that are then applied to the input application source code. Regarding instrumentation and code extensions and modifications, Aspect-Oriented Programming (AOP) approaches, such as AspectC++ [9], can provide some of the required features. A first approach addressing a more flexible approach was presented with the Harmonic+ compiler [10], with the use of the AOP-inspired LARA DSL and the ROSE framework.

More recently, Clava has been proposed as a source-to-source compilation framework for C/C++[1] that allows developers to express analysis and transformation strategies using the LARA DSL [11]. This DSL, which is based on JavaScript, provides a simpler way to perform the code analysis and transformation of previous approaches, and introduces a novel approach for specifying strategies that capture run-time behavior, by enabling compilation and execution of source code from within the same DSL script where transformations are specified.

Although Clava has been used in previous papers, e.g., phase ordering of compiler flags [12], runtime adaptivity and autotuning [13], and automatic parallelization via OpenMP [14], this article is the first publication describing the full framework and to make it available to interested users. In order to clearly show the capabilities and maturity of Clava+LARA, we present new results that where taken after applying Clava over the source-code of a set of large and complex programs (e.g., GCC).

This article is structured as follows. Section 2 presents the framework and its most relevant features, Section 3 shows some illustrative examples and how other researchers have used Clava, Section 4 presents several experiments, including a stress test and a strategy that extracts run-time information, and Section 5 concludes this article.

[1] Clava also supports OpenCL code, and software applications that mix both C/C++ and OpenCL.

2. Software description

Clava is a source-to-source compilation framework for C/C++, built on top of the LARA Framework [11,15], and able to apply strategies written in the LARA language. This section describes the framework, introduces the LARA language, and presents some of the available features.

2.1. Clava framework

The development of Clava started in the beginning of 2016, in the context of the ANTAREX EU project [15] funded by the Horizon 2020 programme, as a way to systematically improve C/C++ applications used in the context of High-Performance Computing (HPC).

Fig. 1 presents three generic use cases for Clava, which have been observed in practice in the work developed by Clava users. In the use case *Reusable Strategies*, the same LARA strategy is applied to several applications. This represents the case where one could encode domain knowledge in a strategy (e.g., loop parallelization [14]). The use case *Custom Targetability* applies different strategies to the same code, in order to obtain different versions of the original code (e.g., parallelize using either OpenMP, or OpenCL). Finally, *Design Exploration* takes advantage of parameterized strategies and applies the same strategy with different parameter values over the same code in a loop, in order to explore different versions of the code (e.g., exploration of variable types with different precisions [16]).

Fig. 2 shows a block diagram of the Clava framework, which is composed by three main parts: 1) the *LARA Engine*; 2) the *Clava Weaver Engine*; and 3) the *C/C++ Frontend*, which contains the *Clava AST Dumper* and the *Clava AST Loader*.

The LARA Framework, written in Java, is focused on the idea that certain tasks and application requirements (e.g., target-dependent optimizations, adaptivity behavior) can be specified separately from the source code that defines the functionality of the program. The framework provides a LARA parser and interpreter (i.e., the *LARA Engine*), which executes LARA scripts that express tasks or strategies to target application requirements. These scripts are then applied to the application source code as a compilation step.

The *Clava Weaver Engine* is responsible for providing information about C/C++, and for maintaining an updated internal representation of the application source code, according to the execution of LARA strategies. The weaver makes the connection between LARA code execution, and the input C/C++ source code.

The C/C++ front-end transforms the source code of the input application into an abstract representation, the Clava Abstract Syntax Tree (AST), which can be manipulated by the Clava Weaver and transformed again into source code. The Clava AST Dumper is a C++ application which uses an unmodified version of Clang, as a library, to parse C/C++ programs. Clang [17] is a production-quality front-end and compiler for languages of the C language family (e.g., C, C++, OpenCL).[2] The Clava AST Dumper parses the source code of C/C++ programs, and generates a dump with syntactic and semantic information about the code, which is then used by the Clava AST Loader to build a Clava AST instance that is equivalent to the original source code. The Clava AST closely resembles the internal AST of Clang, with modifications and extensions that allow AST-based transformations, and the capability of generating source code from the AST. This capability enables AST-based source-to-source, and allows us to perform certain kinds of transformations that with Clang's text-based source-to-source might be impractical (e.g., AST-based function inlining).

[2] Since Clang is used as a parser, Clava can potentially support the same languages as Clang (e.g., CUDA).

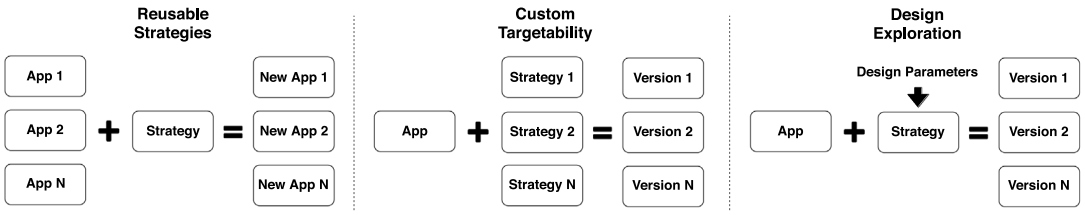

Fig. 1. Generic Clava use cases.

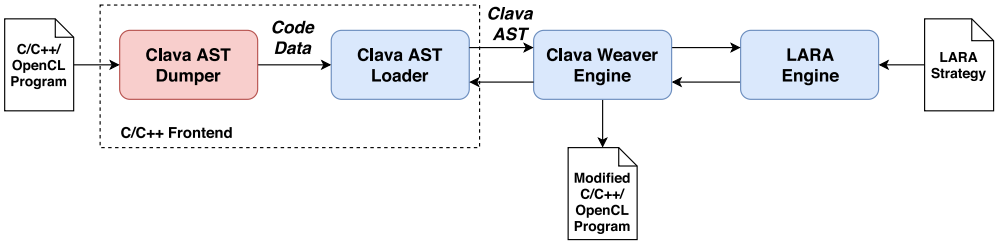

Fig. 2. Diagram of the Clava framework top level structure and compilation flow.

2.2. The lara language

The LARA language [10,11,18] mixes a declarative style, to query the input program code, with an imperative style, that applies actions to each code element (e.g., add monitoring code before the code element). The language extends the ECMAScript 5 specification[3] with several keywords and syntax constructs. The main module of LARA is the `aspect`. Aspects are similar to functions, and is where DSL-specific constructs, such as `select`, can be used. LARA scripts can contain arbitrary JavaScript code, however, DSL constructs cannot be used outside aspects, e.g., in JavaScript functions. Fig. 3 shows an example of an aspect that uses the following LARA keywords: `aspectdef`, `select`, `apply`, and `condition`.

The keyword `aspectdef` (line 1) marks the beginning of an aspect. If a target source code (e.g., a C++ program) has been specified in configuration, before executing the `.lara` file Clava implicitly parses and builds an AST representation that is accessible during the execution of the LARA strategy. The keyword `select` (line 2) allows one to specify the points in the code (e.g., `function`, `loop`) to analyze or transform. The selection is hierarchical and similar to a query, e.g., `select function.loop end` selects all the loops inside all the functions in the target source code.

The `apply` block (lines 3–6) is similar to a `for` loop that iterates over all the points of the previous selection. Each particular point in the code, herein simply referred as *join point*,[4] can be accessed inside the `apply` block by prefixing a dollar sign (i.e., $) to the name of the join point (e.g., `$loop`). Each join point has a set of *attributes*, which can be accessed (e.g., `$loop.line`), and a set of *actions* which can be used to transform or add new code (e.g., `insert` in line 5).

Finally, the `condition` keyword (line 7) can be used to filter join points over a join point selection (resultant from the select statement). Summarizing, the example of Fig. 3 selects all loops inside functions called `foo`, and for each loop, it prints, at source-to-source compilation time, both the line in the source code where the loop is and if it is an innermost loop, and insert the code `'// Before loop'` before the loop.

This example shows that it is possible to express source code modifications in LARA in a compact and clean way. For instance, directly using Clang as a C++ library to implement a similar example can require more than one hundred lines of code, and learn an API that is significantly more complex.[5]

Clava relies on a *Language Specification* that defines the join points available for a given target language (e.g., `function`, `loop`), how they can be selected (e.g., `function.loop`), and which attributes and actions are available for each join point (e.g., `$function.name`).

The Clava Weaver is responsible for defining the Language Specification for C/C++, which is accessed by the LARA Framework during execution of each LARA aspect. The Clava Weaver is also responsible for mapping the join points obtained by a `select` to the equivalent nodes in the AST, and for implementing the attributes and actions (see Fig. 4).

2.3. Clava features

This section presents some features and support tools of the Clava framework.

2.3.1. Installation and platform requirements

The Clava framework is mainly written in Java, with a part written in C++, which uses Clang. The Java code is platform independent and can run in any computer that has a recent Java runtime installed. The Java code can be compiled from scratch using a custom build configuration, or a pre-compiled JAR file can be downloaded.[6]

The C++ code is a standalone executable that needs to be compiled for each platform where one wants to run Clava, and is required for executing the Clava framework. This executable embeds part of Clang as a library, and in order to compile the executable, it is necessary to compile Clang (or have the Clang object-files available) for the target platform. Since this compilation process can be complex and time-consuming, we provide pre-compiled binaries of the stand-alone executable for several platforms. The first time Clava executes, the correct binary is automatically downloaded. Currently, Clava provides pre-compiled

[3] Support for some features of more recent specifications has been added, such as the `for...of` statement.

[4] In the aspect-oriented programming (AOP) terminology they are usually referred as join point shadows [19].

[5] A similar example that uses Clang to insert code can be found here: https://github.com/eliben/llvm-clang-samples/blob/master/src_clang/tooling_sample.cpp.

[6] https://github.com/specs-feup/clava/tree/master/ClavaWeaver.

```
aspectdef ReportInnermostLoops
  select function.loop end
  apply
    println($loop.line + " -> " + $loop.isInnermost);
    $loop.insert before "// Before loop";
  end
  condition $function.name === "foo" end
end
```

Fig. 3. Simple example of a LARA file.

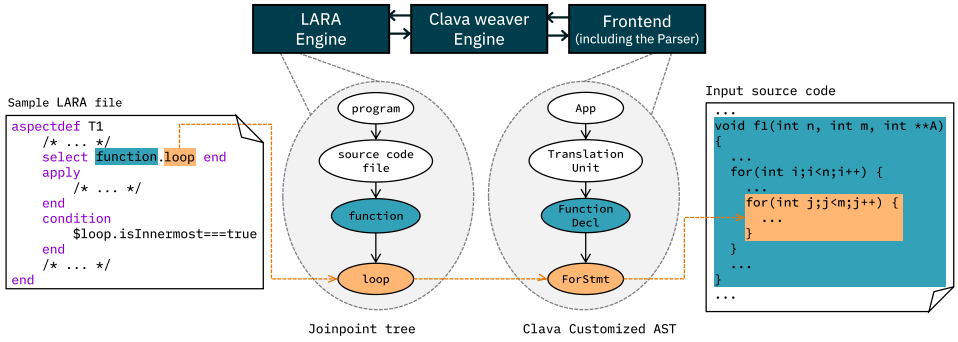

Fig. 4. The mapping between aspects, join points, the Clava AST, and the input source code.

x86 executable binaries for Windows, Ubuntu, CentOS/Fedora and MacOS, and an `ARM V7` executable binary for Ubuntu. For Linux platforms, we provide `clava-update`, an installation and update script.

The Clava framework currently provides three tools, `clava` (the weaver), `clava-doc` (see Section 2.3.3) and `clava-unit` (see Section 2.3.4). Besides these tools, it also installs CMake[7] modules for Clava (see Section 2.3.2).

Finally, we have developed an online version of Clava with several examples, where users can immediately test the tool on a limited context.[8] This online version is interesting for preliminary user experiences while deciding to opt for the installation of the full framework.

2.3.2. CMake integration

As CMake [3] is a popular tool for build management that supports C/C++ compilation, the Clava framework provides integration through a CMake module[9] which enables the application of LARA scripts on the source code tree. This module allows to specify custom analysis and transformation steps that are applied just before the code is compiled.

The module package, `ClavaConfig`, checks for Clava dependencies, downloads the Clava weaver executable if not installed, and adds new functions to CMake (e.g., `clava_weave`).

2.3.3. Documentation generator

We provide a standalone documentation generator for the LARA DSL, *LaraDoc*. It accepts LARA code as input, and generates an HTML document based on the annotations in the source files. We adopted JSDoc annotations[10] and added LARA-specific annotations (e.g., `@aspect`).

LaraDoc is part of the Lara Framework, and is independent from Clava. We added the utility *clava-doc*, which generates documentation taking into account Clava APIs. The Clava documentation for the Clava Standard Library is generated using this tool and is available online (see Section 2.4).

2.3.4. Testing framework

Clava includes a tool for performing unit testing. Fig. 5 shows a LARA file that contains two unit tests. A LARA unit test is a LARA aspect or function annotated with the tag `@test` in the comment that precedes the aspect or function. A test passes if no exception is thrown during execution of the test, and fails otherwise (i.e., `TestFail()`). After execution, `clava-unit` generates a report (see Fig. 6).

2.3.5. Data annotated in source code

It is possible to communicate information from the source code to Clava by using the pragma `clava data`, in the source code, and the attribute `data`, in LARA. Fig. 7 shows an example where we use this pragma to annotate a function and define two key/value pairs, with keys `aNumber` and `aString`, and values `0` and `'aString'`, respectively. The values can be any valid LARA code. After annotating the source code, this information can be accessed using the attribute `data`, as shown in Fig. 8.

2.4. Built-in actions and clava API

Clava provides built-in actions associated to certain join points, which can be called directly from LARA code. Table 1 shows some examples of the built-in actions. Some actions are available to all join points (e.g., `copy`), but generally, each join point has its own set of actions (e.g., `inline` for calls, `interchange` for loops). A complete list of supported actions is available online.[11]

LARA also supports the `import` keyword, in order to enable basic modularity for LARA strategies. For example, we have used the import feature to develop a standard library for the LARA framework and Clava. Table 2 shows some examples of the APIs provided by Clava. The row LARA refers to examples of classes that are agnostic to the target language, and that are available in other LARA compilers. The row Clava are examples of classes that are exclusive to Clava, usually related to C/C++ source-to-source tasks. Comprehensive documentation for the Clava API is available online.[12]

7 https://cmake.org/.
8 http://specs.fe.up.pt/tools/clava/.
9 https://github.com/specs-feup/clava/tree/master/CMake.
10 http://usejsdoc.org/.
11 http://specs.fe.up.pt/tools/clava/language_specification.html.
12 http://specs.fe.up.pt/tools/clava/doc/.

```
import Foo;

/**
 * A test aspect that calls Foo.
 * @test
 */
aspectdef TestFoo
  call Foo();
end

/**
 * A test function that will fail.
 * @test
 */
function TestFail() {
  throw "This test failed!";
}
```

Fig. 5. LARA test file with two unit tests.

```
LaraUnit test report

Failed tests:
 - FooTest.lara::TestFail (2.44s)
 [ERROR] User exception on line 16: This test failed!
 [STACK]
 During LARA Interpreter execution
  caused by RuntimeException:

   Main@/home/user/LaraUnitTestFolder/test_function.lara, line 4
    TestFail@/home/user/LaraUnitTestFolder/test_function.lara, line 6
     User exception on line 16: This test failed!


Passed tests:
 - FooTest.lara::TestFoo (4.53s)

Total Tests: 2
Passed / Total: 1 / 2
Failed: 1

 SOME TESTS FAILED
```

Fig. 6. Example of a `clava-unit` test report.

```
#pragma clava data aNumber:0, aString:'aString'
void foo();
```

Fig. 7. Example of C/C++ source code annotated with the pragma `clava data`.

```
select function{"foo"} end
apply
  println("Should be the number 0: " + $function.data.aNumber);
  println("Should be string aString: " + $function.data.aString);
end
```

Fig. 8. Example of LARA code that uses the attribute `data`.

2.5. Miscellaneous features

Clava provides the following miscellaneous features:

- *Parallel Parsing* — Clava supports parsing the source files in parallel. This allows us to considerably speedup parsing time for projects that contains a large number of source files;
- *Mixed C/C++ and OpenCL projects* — it is possible to include C/C++ and OpenCL files in the same Clava compilation project. In this case, all files become part of the AST, and it is possible to write strategies that require information about the different kinds of files (e.g., change the parameters of an OpenCL kernel and adapt the corresponding call in the C/C++ host code);

3. Illustrative examples

This section presents a set of examples of LARA strategies that can be executed by Clava, as well as several use cases where Clava has been used to solve specific problems.

3.1. Instrumentation

Instrumentation is a common use case for source-to-source compilers, and can be used to collect runtime data, or add logging information.

Table 1
Example of Clava built-in actions.

Join Point	Action	Description
All	*detach*	Removes node from the tree
	copy	Creates an out-of-the-tree copy of this node
	setValue	Sets the value of a given property of the node
$program	*rebuild*	Reparses current AST, transforming literal code into AST nodes
	push	Copies the current AST and pushes it to the top of the AST stack
	pop	Discards the AST at the top of the AST stack
$file	*rebuild*	Reparses this file
	addGlobal	Adds a global variable to this file
	write	Writes the code of this file to a given folder
$call	*wrap*	Creates new function with this call, replaces current call with call to wrap function
	inline	Inlines this call
$function	*clone*	Inserts a renamed copy of this function in the program
	newCall	Creates a new call node to this function
$loop	*interchange*	Applies loop interchange between this loop and a second given loop
	tile	Applies loop tiling to this loop

Table 2
Example of Clava API.

API	Class	Description
LARA	Io	Basic I/O functionality related to files
	Platforms	Information about the platform that is executing the compiler (e.g., is Linux?)
	CMaker	Compiles current AST into an executable
	ProcessExecutor	Launches command-line processes
Clava	Parallelize	Auto-parallelization strategies for loops, using OpenMP [14]
	Gprofer	Compiles current AST and runs executable with gprof profiler
	Hdf5	Automatically generates HDF5 wrappers for C++ structures and classes [20]

Fig. 9 shows LARA code to instrument outermost loops in order to log when they are executed, and obtain runtime information about how long they take to execute and how much energy they consume. Lines 1–3 import three classes, `Logger`, `Timer` and `Energy`, which are instantiated in lines 7–9. Line 11 selects all the loops that are outermost (i.e., loops that are not inside another loop), and for each loop, creates an `id` based on its file, function and rank (line 13), inserts before the loop code that will print a message, during program execution, when the loop is executed (line 14), and measures the execution time and energy consumption of the loop, using the loop *id* as a tag (lines 15–16).

3.2. Code transformations

It is possible to improve certain properties of an application (e.g., execution time) by applying transformations to its source code. Fig. 10 shows a LARA script that performs loop interchange, i.e., exchanges the order of two nested loops. This technique can be used to ensure that memory accesses are done in an order that is closer to how the data is laid out in memory, improving data access locality.

Line 7 selects two loops that are part of a loop nest and assigns a name to each of them (`$lo` and `$li`). If the loops are inside a function with the given name, and have the expected control variables (lines 9–11), interchange is performed (line 12). Since the LARA code accepts input parameters, it can be used as a building block for more complex LARA aspects, e.g., automatically exploring several interchange combinations of a program's loops.

3.3. Use cases

Clava has been designed to be a very accessible code analysis and transformation framework for C/C++, and to have a large application spectrum. It has already been used as a contributing framework for scientific research achievements, such as:

- exploration of phase ordering and phase selection of compiler flags [12]
- automatic parallelization of `for` loops with OpenMP pragmas [14]
- performance tuning of High Performance Computing (HPC) applications [13]
- mixed-precision tuning for OpenCL kernels on GPUs [16]
- automatic code generation for HDF5 serialization [20]

Several students have also used or are currently using Clava in class assignments and internships, and have explored topics such as memory layout optimization, extraction of code features, performance modeling,[13] and design-space exploration.[14]

4. Impact

We believe that Clava can be a useful compiler framework for a wide range of C/C++ users and developers, from experts that can write their own custom LARA scripts, to users that can easily apply existing LARA scripts (e.g., with the CMake plug-in).

In order to briefly show the impact of the Clava compiler, we have prepared three examples: a stress-test that processes large programs; a demonstration of an interface that helps performing reproducible experiments with Clava; and an experiment that transforms, compiles and executes source code from within Clava.

All experiments where done on a computer running Ubuntu 16.04 LTS (64-bit), with two Intel Xeon E5-2630 v3 CPUs (@2.40 GHz) and 128 GB of DDR4 RAM. The programs were compiled with GCC 7.4.0 with `-O3`.

4.1. Stress test

In this section we exercise several Clava capabilities, such as parsing, instrumentation and code generation. In order to provide a stress test for Clava, we have collected some applications with large amounts of lines of code.

We used benchmarks from two sources, the NAS Parallel Benchmarks [21], and Stephen McCamant's *Large single compilation-unit C programs*.[15] For the experiments, we automatically instrument the source code of each benchmark to produce a Dynamic Call Graph (DCG) at runtime. The execution times presented are an average of 30 executions after discarding the first result, in order to warm-up the Java Virtual Machine. The examples are available online[16] and it is possible to replicate all results by calling a single Clava command.

Table 3 presents a characterization of the benchmarks, and information about Clava execution and about the graphs generated by the applications after instrumentation. The first column, *C Code Lines*, presents the size of each program, in total lines, including comments and blank lines. The great majority of the benchmarks have more than one thousand lines, and the largest one, `gcc`, has more than 750,000 lines. To present a measure of the complexity of the program that is not dependent on how the code is formatted, we have included the column *AST Nodes*, which

13 https://github.com/joseloc300/AutoParSelector.
14 https://github.com/specs-feup/LAT-Lara-Autotuning-Tool.
15 http://people.csail.mit.edu/smcc/projects/single-file-programs/.
16 https://github.com/specs-feup/clava-examples/tree/master/2019_Stress_Test.

```
import lara.code.Logger;
import lara.code.Timer;
import lara.code.Energy;

aspectdef InstrumentLoops

    var logger = new Logger();
    var timer = new Timer();
    var energy = new Energy();

    select file.function.loop{isOutermost === true} end
    apply
        var id = $file.name + "_" + $function.name + "_" + $loop.rank;
        logger.append("Executing loop " + id).ln().logBefore($loop);
        timer.time($loop, "time " + id);
        energy.measure($loop, "energy " + id);
    end

end
```

Fig. 9. LARA script for code instrumentation.

```
aspectdef LoopInterchange
    input
        functionName = 'matrix_mult',
        outerIndex = 'i', innerIndex = 'j'
    end

  select function.($lo = loop).($li = loop) end
  apply
    if($function.name === functionName &&
    $lo.controlVar === outerIndex &&
    $li.controlVar == innerIndex) {
        $lo.interchange($li);
    }

  end
end
```

Fig. 10. Parameterizable LARA script for loop interchange.

shows the number of nodes in the Clava AST after parsing the code.

All benchmarks were successfully parsed by Clava, including the largest application, which after parsing generated an AST that contained more than 2 million nodes. After instrumentation and code generation from the AST, all benchmarks successfully compiled and executed. Upon execution, each benchmark produced a graph in `dot` format with the corresponding dynamic call graph of the execution.

Column *Frontend* shows the time it took to parse the input code and load the AST (i.e., *Clava AST Dumper* and *Clava AST Loader* in Fig. 2, respectively), and column *Weaving* shows the time it took to instrument the code in order to produce a dynamic call graph. In most cases, the number of lines of code inserted by Clava (i.e., column *C Lines Diff*) ranges from hundreds to thousands of lines. For the largest applications, doing this kind of instrumentation by hand would be impractical. The execution time seems to increase linearly with the number of AST nodes, and in all cases, the parsing time dominates the instrumentation time, which is under 1 s for all examples except for `GCC`. The purpose of this experiment was to demonstrate the efficiency and robustness of the Clava framework. The instrumented versions of all examples generated a dynamic call graphs upon execution, with graph sizes that ranged from around a dozen nodes to almost 200 nodes.

4.2. Benchmark packaging

Analyzing, transforming and testing a set of benchmarks may require non-trivial setup effort (e.g., defining inputs, compiler configurations, machine setup, file organization) that can be hard to correctly replicate.

The Clava framework provides the interfaces `BenchmarkSet` and `BenchmarkInstance`, which enable packaging a set of benchmarks so that the laborious process that was previously mentioned can be replaced by a single `import` and a simple API. Clava already has support for specifying, in LARA scripts, how a program should be loaded, parsed, compiled and executed. Benchmark packaging raises the abstraction level one step further.

Fig. 11 shows a LARA example that uses a packaged version of the NAS benchmark.[17] The example contains two aspects, `NasTest`, which is shown here just to demonstrate the internal execution flow, and `NasTestSimpler`, which presents a simpler way to use a benchmark set.

First, the benchmark set is imported (line 1) and instantiated (line 5). A benchmark set represents a set of benchmarks and corresponding inputs, which can be configured (lines 6–7). After configuration, one can obtain a list of all benchmarks-inputs combinations that will be executed (i.e., function `getInstances()` and iterate over the benchmark instances (lines 9–15). After loading a benchmark instance (line 10), the code becomes available as an AST in Clava, and one can apply any code analysis or transformation (line 11). To perform an analysis that requires run-time information, it is possible to compile (line 12) and execute (line 13) the benchmark, and parse its outputs. Before loading the next benchmark instance, one should close the current instance (line

[17] https://github.com/specs-feup/clava/tree/master/benchmarks/NAS.

Table 3
Results of C Stress test.

Benchmark	Original Code		Clava Execution			Call Graph (Generated)	
	C Code Lines	AST Nodes	C Lines Diff	Frontend (ms)	Weaving (ms)	Nodes	Edges
NAS_EP	571	1621	107	562	10	17	20
NAS_IS	777	1696	86	579	9	14	15
NAS_FT	1151	4672	154	867	16	29	34
NAS_CG	1267	3561	128	675	14	20	23
NAS_MG	1749	7418	225	947	24	27	37
NAS_SP	3165	23887	186	2010	75	31	40
NAS_LU	3742	27509	214	2300	101	31	37
NAS_BT	3798	32405	213	2504	116	30	46
bzip2	6998	34817	1508	2631	146	78	103
gzip	8607	23944	1454	1858	123	76	99
oggenc	58413	323144	4305	29413	735	21	21
gcc	753820	2630805	184632	177519	14641	193	276

14) to discard the benchmark and restore the original state of the AST.

`BenchmarkSet` implements a generator function that performs the needed house-keeping tasks when iterating over benchmark instances, so when we use an instance of a benchmark set directly in a `for...of` (line 21), we do not need to worry about loading and closing benchmark instances. The function `execute()` also takes care of compiling the code, if it has not been compiled yet.

Benchmark instances also provide the function `getKernel()`, which returns the region in benchmark code where the main computation is performed (e.g., after initialization and warm-up, before results processing) and that we usually want to measure. In the case of the NAS bencharmks, in order to identify the kernel, we manually added a `#pragma kernel` before regions that where delimited by timers that where originally in the benchmark. If timers where between a `for` loop, the pragma was inserted directly before the loop; otherwise, if the timers where between a list of statements, we created a new scope and inserted the pragma before the scope.

4.3. Executed operations in a code region

We show in this example the capabilities of Clava to instrument and test code. Specifically, we instrument, compile and execute the NAS benchmarks, in order to calculate, for each kernel and for a given input, a reproducible estimation of the number of executed operations. Such an estimation might be useful to analyze an application and may complement profiling results. We note, however, that an exact measure of the actual executed operations by a given processor would require counting operations at the processor level, and is highly dependent on, among other things, the compiler, which compilation flags where used, and the processor where the program was executed.

Fig. 12 shows a LARA script that implements the complete example. As in the previous example, the script starts with the import of the NAS benchmark set (line 2), creates a new instance (line 7), and configures it (line 8). Then the LARA script iterates over all benchmark instances (line 9).

For each benchmark instance, a new `OpsCounter` is created, which is used to instrument the code of the benchmark to count the operations inside the benchmark kernel (line 11), and to print the results after the execution of the kernel (line 12). Then, the benchmark is compiled and executed (line 13), and the output is saved to a file (lines 14–15) for later processing.

This simple script allows to obtain the information presented in Table 4. It shows the number of executed operations in the kernels of the NAS benchmarks, when running with sizes W and A. According to the results, size A requires an order of magnitude more operations than size W. In addition, the strategy provides information about each operation type (e.g., floating-point vs integer) and bit-width (e.g., 32-bit vs 64-bit). As expected, NAS benchmarks are floating-point intensive codes, although in some cases, integer operations represent a significant part of the total (e.g., IS, MG). Note that the total of operations for the benchmark *NAS_IS* is not the sum of the integer and floating-point operations as there are pointer-based arithmetic operations not explicitly represented as a column of Table 4.

Table 4
Operations executed by the kernels in NAS benchmarks.

Benchmark	Total MOps		MIops (32 bits)		MFlops (64 bits)	
	Size W	Size A	Size W	Size A	Size W	Size A
NAS_BT	9109	199641	1187	26581	7922	173059
NAS_CG	452	1600	0	0	452	1600
NAS_EP	1594	12752	101	805	1493	11946
NAS_FT	914	17306	236	4307	678	12999
NAS_IS	13	104	1	6	0	0
NAS_LU	26405	176137	4582	30640	21823	145498
NAS_MG	1497	11880	820	6491	677	5389
NAS_SP	17114	103519	3197	19435	13918	84084

As an example of Clava flexibility, we can use the execution time measurement presented in Section 3.1 to extend the operation counting script and calculate an estimation of the kernel operations per second (since counting operations can introduce overhead, the execution time should be measured in a separate run, without instrumentation for counting operations).

Users can also copy the OpsCounter file[18] and develop their own operation counting metric, for instance, by defining which operations to count, or by attributing custom weights to each operation.

5. Conclusion

This article presented Clava, a source-to-source compilation framework for C/C++. The focus of the Clava framework is to provide a programming environment that has a wide range of applicability, as well as features that lower the entry barrier for new users and contribute to a better user experience. The framework provides an enhanced source-to-source compilation approach that offers a user interface to strategies based on the LARA framework. With Clava+LARA, users can program their own strategies and automatically apply scripts that analyze, instrument, transform, compile and execute a given source-code, all within the same script. In addition, the framework provides a set of libraries that can be useful as building blocks for programming new strategies. We presented the framework, its structure and main tools, the principles behind its design, and a number of simple but representative examples that show some of the features of the Clava+LARA approach.

[18] https://github.com/specs-feup/clava/blob/master/ClavaLaraApi/src-lara-clava/clava/clava/stats/OpsCounter.lara.

```
import lara.benchmark.NasBenchmarkSet;

aspectdef NasTest

    var benchSet = new NasBenchmarkSet();
    benchSet.setBenchmarks("CG", "EP");
    benchSet.setInputSizes("S", "W", "A");

    for(var bench of benchSet.getInstances()) {
        bench.load(); // Replaces current AST with benchmark's AST
        // ... transformations and analysis
        var executable = bench.compile();  // Compile benchmark
        var processExecutor = bench.execute(); // Execute benchmark
        bench.close(); // Unloads the benchmark and restores previous AST
    }
end

aspectdef NasTestSimpler

    // ... benchSet initialization
    for(var bench of benchSet) {
        // ... transformations and analysis
        var processExecutor = bench.execute(); // Execute benchmark
    }
end
```

Fig. 11. Example of LARA code that uses a packaged version of the benchmark NAS.

```
import clava.stats.OpsCounter;
import lara.benchmark.NasBenchmarkSet;
import lara.Io;

aspectdef OpsCounterTest

    var benchSet = new NasBenchmarkSet();
    benchSet.setInputSizes("W", "A");
    for(var bench of benchSet) {
        var opsCounter = new OpsCounter();
        opsCounter.instrument(bench.getKernel());
        opsCounter.log(bench.getKernel());
        var process = bench.execute();
        var filename = process.getExecutableFile().getName()+".txt";
        Io.writeFile(filename, process.getConsoleOutput());
    }
end
```

Fig. 12. Example of LARA code that instruments and runs the NAS benchmarks in order to estimate the number of executed operations.

As future work, we have plans to use and evaluate Clava in projects involving topics such as detection and patching of code vulnerabilities in C/C++, advanced custom precision strategies, implementation of software metrics, obtaining AST-based code features for machine learning, and moving computations from C to hardware accelerators.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by the ANTAREX project through the EU H2020 FET-HPC program under grant no. 671623. João Bispo acknowledges the support provided by Fundação para a Ciência e a Tecnologia, Portugal, under Post-Doctoral grant SFRH/BPD/118211/2016.

References

[1] Chung L, Nixon BA, Yu E, Mylopoulos J. Non-functional requirements in software engineering, vol. 5. Springer Science & Business Media; 2012.
[2] Muschko B. Gradle in action. Manning; 2014.
[3] Martin K, Hoffman B. Mastering CMake: a cross-platform build system. Kitware; 2010.
[4] Dave C, Bae H, Min S-J, Lee S, Eigenmann R, Midkiff S. Cetus: A source-to-source compiler infrastructure for multicores. Computer 2009;42(12):36–42.
[5] Quinlan D. ROSE: Compiler support for object-oriented frameworks. Parallel Process Lett 2000;10(2–3):215–26.
[6] Irigoin F, Jouvelot P, Triolet R. Semantical interprocedural parallelization: An overview of the PIPS project. In: ICS, vol. 91. 1991, p. 244–51.
[7] Insieme compiler and runtime infrastructure, Institute of Computer Science, Distributed and Parallel Systems Group, University of Innsbruck, http://insieme-compiler.org Retrieved: 08-06-2019.
[8] Chen C, Chame J, Hall M. Chill: A framework for composing high-level loop transformations. In: Tech. Rep. 08-897. U. of Southern California; 2008, p. 136–50.
[9] Spinczyk O, Gal A, Schröder-Preikschat W. Aspectc++: An aspect-oriented extension to the c++ programming language. In: Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, CRPIT '02. Darlinghurst, Australia, Australia: Australian Computer Society, Inc.; 2002, p. 53–60.
[10] Cardoso JMP, Coutinho JG, Carvalho T, Diniz PC, Petrov Z, Luk W, Gonçalves F. Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach. Softw - Pract Exp 2016;46(2):251–87.
[11] Pinto P, Carvalho T, Bispo J, Ramalho MA, Cardoso JMP. Aspect composition for multiple target languages using LARA. Comput Lang Syst Struct 2018;53:1–26.
[12] Nobre R, Bispo J, Carvalho T, Cardoso JM. Nonio – modular automatic compiler phase selection and ordering specialization framework for modern compilers. SoftwareX 2019;10:100238.

[13] Pinto P, Bispo J, Cardoso JMP, Barbosa JG, Gadioli D, Palermo G, Martinovic J, Golasowski M, Slaninova K, Cmar R, Silvano C. Pegasus: performance engineering for software applications targeting hpc systems. IEEE Transactions on Software Engineering 2020. 1–1.

[14] Arabnejad H, Bispo J, Cardoso JM, Barbosa JG. Source-to-source compilation targeting OpenMP-based automatic parallelization of C applications. J Supercomput 2019;1–33.

[15] Silvano C, Agosta G, Bartolini A, Beccari AR, Benini L, Besnard L, ao Bispo J, Cmar R, ao M.P. Cardoso J, Cavazzoni C, Cesarini D, Cherubin S, Ficarelli F, Gadioli D, Golasowski M, Libri A, Martinovi J, Palermo G, Pinto P, Rohou E, rina Slaninová K, Vitali E. The ANTAREX domain specific language for high performance computing. Microprocess Microsyst 2019;68:58–73.

[16] Nobre R, Reis L, Bispo J, Carvalho T, Cardoso JM, Cherubin S, Agosta G. Aspect-driven mixed-precision tuning targeting gpus. In: Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms. ACM; 2018, p. 26–31.

[17] clang: a C language family frontend for LLVM, Retrieved: 15-09-2018 http://clang.llvm.org/.

[18] Cardoso JM, Carvalho T, Coutinho JG, Luk W, Nobre R, Diniz P, Petrov Z. LARA: an aspect-oriented programming language for embedded systems. In: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development, 2012, pp. 179–190.

[19] Fabry J, Kellens A, Ducasse S. Aspectmaps: A scalable visualization of join point shadows. In: 2011 IEEE 19th International Conference on Program Comprehension. IEEE; 2011, p. 121–30.

[20] Golasowski M, Bispo J, Martinovič J, Slaninová K, Cardoso JMP. Expressing and applying c++ code transformations for the hdf5 api through a DSL. In: Saeed K, Homenda W, Chaki R, editors. Computer Information Systems and Industrial Management. Cham: Springer International Publishing; 2017, p. 303–14.

[21] Bailey DH. NAS Parallel benchmarks. In: Encyclopedia of parallel computing. Springer; 2011, p. 1254–9.