Chapter 2. DSL and Source to Source Compilation: the Clava+LARA approach

João Bispo, Pedro Pinto, and João M.P. Cardoso

Faculty of Engineering, University of Porto, Porto, Portugal
{jbispo@fe.up.pt, p.pinto@fe.up.pt, jmpc@fe.up.pt

Abstract

Clava is a source to source (C++ to C++) compiler entirely developed during the ANTAREX project. It includes an aspect-oriented programming approach, implemented by an internal weaver and the technology provided by the LARA DSL, in order to describe source-to-source strategies, such as code transformations and instrumentation. In most cases, the strategies are applied offline and/or translated to code in the target programming language and embedded in the application code. The version of Clava presented in this chapter is able to compile a wide range of applications, and several kinds of strategies have been written in the ANTAREX DSL, including auto-parallelization, design space exploration, source-code generation and automatic integration of other ANTAREX libraries and tools. In addition to the capability for code transformations, code instrumentation, and code injection for integration of runtime autotuning and adaptivity schemes, Clava also includes data dependence analysis stages that are used by the autoparallelizer via OpenMP directives. This chapter presents the Clava compiler how the interaction with LARA works and includes a number of examples (with all software code available) showing some of the advantages and usefulness of the approach.

1 Introduction and Motivation

Trade-offs are, arguably, one of the cornerstones of engineering, and when writing software there are many kinds of trade-offs that can be done. For instance, code can be written to be more readable, or more efficient; to use less memory, or computation; be more generic, or specialized to a given platform. There is no "unique" correct implementation for a piece of code, and is instead highly contingent on the non-functional requirements [1] and target platforms.

Heterogeneous architectures include, in the same system, different targets (e.g., CPUs, GPUs, FPGAs) which might be better suited to certain jobs than others, according to a given metric (e.g., execution time, power and energy consumption). While these architectures have been the norm in domains such as Embedded Computing, problems such as power dissipation [2] have been driving the adoption of heterogeneous architectures in other domains (e.g., datacenters [3], High-Performance Computing (HPC) [4]). Heterogeneous architectures further exacerbate the problem of writing software that fully takes advantage of the target platform. Writing code that efficiently takes advantage of a single target (e.g., Intel CPU) is not easy and requires expert domain knowledge. Having several different targets in the same system makes the problem even more complex.

C and C++ are programming languages that favor performance and efficiency over other characteristics, such as compilation time, or convenience [5]. They are relatively low-level and close to the hardware, and usually allow fine-grained control of system resources (e.g., memory, co-processors). C and C++ are still widely used to program platforms in fields such as Embedded Computing and HPC. However, due to the focus these languages have in performance, they may lack features that are commonplace in other widely used languages (e.g., garbage collection, comprehensive reflection support).

There have been many efforts to manage the complexity of programming heterogeneous systems. A common and successful approach is to develop libraries and compiler-supported extensions for a given language (e.g., OpenMP [6] and Threading Building Blocks [7], for C/C++). Libraries and language extensions can encapsulate complex functionality around interfaces that can greatly help to take advantage of a platform. However, it can be non-trivial to correctly apply them. For instance, it is relatively common for inexperienced users to obtain *worse* performance from an application when naively applying a parallelization framework such as OpenMP[1]. Also, changes related to performance and efficiency usually require a profiling step as well as several test steps, and activity where the code is manually changed and that can be error-prone and time consuming.

Another approach is to develop a new language that specifically handles a certain problem or set of problems (Domain Specific Languages, or DSLs). An advantage of DSLs over previous approaches 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 the introduction of errors, or inefficient idioms. On the downside, DSLs introduce the overhead 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.

DSLs can be very general, or incredibly specific. OpenCL [8] is an example of a DSL which can express general-purpose computation, as is used to write programs that are functionally portable across heterogeneous platforms (although not performance portable [9]). On the other hand, CHiLL [10] is a declarative language that specifies sequences of (predefined) transformations and constrains that are to be applied to existing source-code.

This chapter presents Clava, a source-to-source compilation framework for C/C++[2] that allows to express analysis and transformation strategies using LARA [11], a JavaScript-based DSL for source code analysis and manipulation.

With this framework, we intend to provide the necessary tools for domain experts to be able to encode their knowledge in reusable strategies for C/C++, written in LARA. Section 2 presents the framework, how it is organized and how it was implemented.

Furthermore, we developed and extended several support tools for the scripting language LARA, such as a documentation generator, a unit testing framework and a standard library. Section 3 presents this work.

Clava was developed to be flexible enough to allow a wide-range of solutions, from code generation to design-space exploration. Section 4 describes use cases from several Clava users. Section 5 presents several approaches that either inspired or are similar to what Clava does, and Section 6 concludes this chapter.

[1]Parallelization usually introduces overhead, due to the creation of additional processes/threads and the synchronization between them.

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

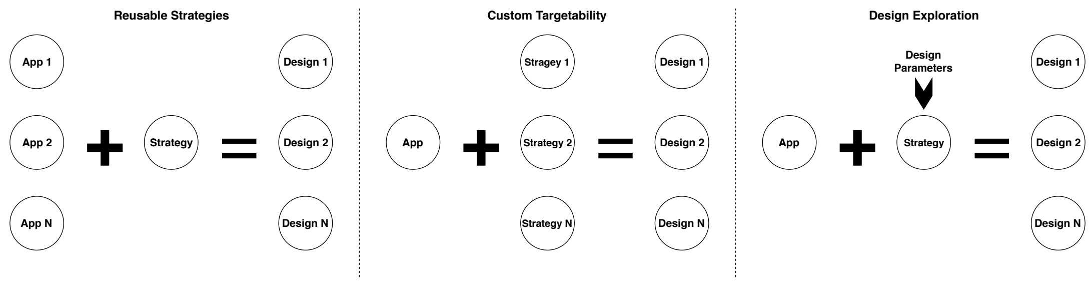


Figure 1: Generic Clava use cases.

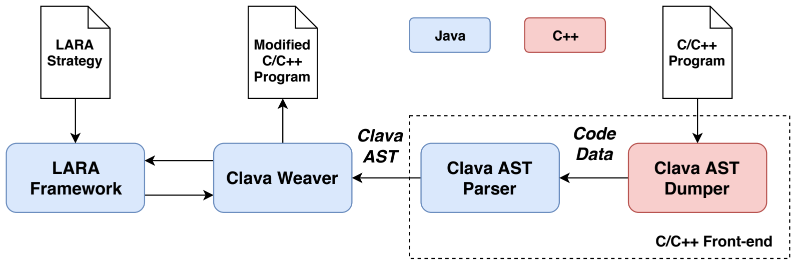


Figure 2: Diagram of the structure of the Clava framework.

2 Clava Framework

Clava is a source-to-source compilation framework for C/C++, built on top of the LARA Framework. Clava parses C/C++ code and applies over it strategies written in the LARA language, a JavaScript-based language for source-code analysis and transformations.

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

Figure 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 programs. This represents the case where one could encode domain knowledge in a strategy (e.g., see loop parallelization in Section 4.2). 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 OpenMP or OpenCL - see Section 4.3). *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., see LAT in Section 4.1).

Figure 2 shows a block diagram of the Clava framework, which is composed by three main parts: 1) the *LARA Framework*; 2) the *Clava Weaver*; and 3) the *C/C++ Front-end*, which contains the *Clava AST Dumper* and the *Clava AST Parser*.

The LARA Framework is a Java library inspired by the Aspect-Oriented paradigm [12], which is based 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 compiler and interpreter for the LARA language, which is used to write strategies that express tasks or application requirements. These strategies are then applied to the user source-code as a compilation step.

The Clava Weaver 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 execu-

Table 1: Number of source code files and lines of source code for each of the projects composing the Clava framework.

Project	No. of Files	SLOC
ClavaAstDumper	39	3095
ClavaAstParser	451	14492
ClavaAst	387	10813
ClavaLaraApi	8	99
ClavaTester	37	3371
ClavaViewer	9	253
ClavaWeaver	189	6052
Total:	**1120**	**38175**

tion, 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 AST (Abstract Syntax Tree), which can be manipulated by the Clava Weaver and transformed again into source code. The Clava AST Dumper is a C++ application which uses Clang [13] as a library, to parse C/C++ programs. Clang is a production-quality front-end and compiler for languages in the C language family (e.g., C, C++, OpenCL, CUDA)[3]. The Clava AST Dumper parses the source code of a C/C++ program, and generates a dump with syntactic and semantic information about the code, which is then used by the Clava AST Parser to create a Clava AST that is equivalent to the original source code. The Clava AST closely resembles the internal AST of Clang, but has modifications and extensions that allow AST-based transformations, and the capability of generating source code from the AST.

2.1 Clava Framework Characterization

Table 1 shows the number of Java source files, as well as the logical lines of source code (excluding comments) in each of the projects that compose Clava. On September 2018, the Clava framewrok consists of 38,175 lines of code written over 1120 files (lines for automatically generated code, LARA source files, and source code that is part of the LARA framework are not included).

The most important Clava projects are the C/C++ front-end (i.e., `ClavaAstDumper` and `ClavaAstParser`), the AST (i.e., `ClavaAst`) and the Weaver (i.e., `ClavaWeaver`). `ClavaAstDumper` is the executable written in C++ that uses Clang to parse the code. `ClavaAstParser` translates the dump of the executable into the AST provided by `ClavaAst`. `ClavaWeaver` is the implementation of the weaver interface provided by the LARA framework, which makes the connection between the Clava AST and the LARA strategies.

2.2 The Lara Language

The LARA language is fully compatible with the ECMAScript 5 specification[4]. This means that any JavaScript code that conforms to that specification is considered valid LARA code.

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

[4]Support for some features of more recent specifications has been added, such as the `for...of` statement, when used in `.lara` files.

```
1 aspectdef HelloWorld
2   select function.loop end
3   apply
4     println($loop.line + " -> " + $loop.isInnermost);
5     $loop.insert before "// Before loop";
6   end
7   condition $function.name === "foo" end
8 end
```

Figure 3: Simple example of a LARA file.

Besides supporting plain JavaScript in `.js` files, LARA extends JavaScript with several new keywords, and syntax constructs, which must be used in `.lara` files. Figure 3 shows an example of LARA code that uses some LARA keywords, `aspectdef`, `select`, `apply` and `condition`.

The keyword `aspectdef` (line 1) marks the beginning of an aspect. The weaver, before execution of a `.lara` file, implicitly parses the target source code (e.g., a C++ program) and builds an abstract representation that is accessible during the execution of the LARA strategy. The keyword `select` (line 2) allows to specify the points in the code (e.g., `function`, `loop`) that we want 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*, 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., `$loop.isInnerMost`), and a set of *actions* which can be used to transform the code (e.g., `$loop.insert before "//Comment before loop"`).

Finally, the `condition` keyword (line 7) can be used to filter join points over a join point selection. Summarizing, the example of Figure 3 selects all loops inside functions called `foo`, and for each loop, it will print both the line in the source code where the loop is and if it is innermost, and insert the code `'// Before loop'` before the loop.

Since LARA is agnostic to the target language (e.g., C/C++), it relies on the weaver to provide a *Language Specification* which specifies which join points are 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`).

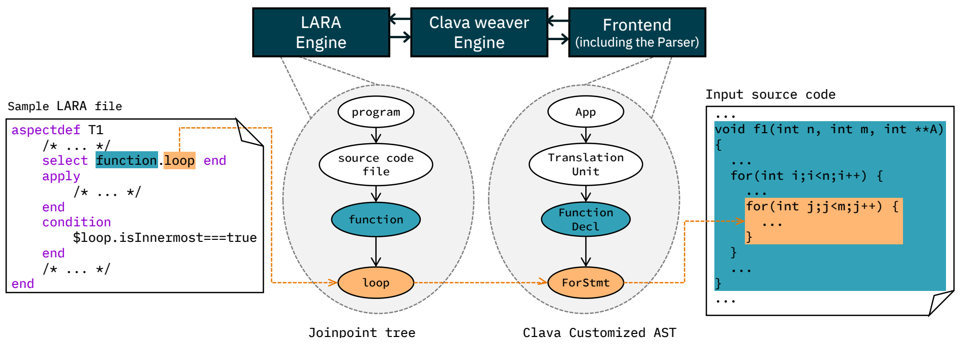
The Clava Weaver is responsible for defining the Language Specification for C/C++, which is accessed by the LARA Framework during execution of the LARA aspect. The Clava Weaver is also responsible for mapping the join points obtained by a `select` to the equivalent nodes in the AST, and for implementing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

3 Clava Features

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

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 part is platform independent and can run in any system that has a Java 8+ runtime



```
aspectdef T1
    /* ... */
    select function.loop end
    apply
        /* ... */
    end
    condition
        $loop.isInnermost===true
    end
    /* ... */
end
```

```
...
void f1(int n, int m, int **A)
{
  ...
  for(int i;i<n;i++) {
    ...
    for(int j;j<m;j++) {
      ...
    }
  }
  ...
}
...
```

Figure 4: The mapping strategy between aspects, join points, the Clava AST, and the input source code

installed. The Java code can be compiled from scratch using our custom build configuration, or a pre-compiled JAR file can be downloaded.

The C++ part 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 platform we are compiling to. Since compiling Clang can be a complex process (compilation can take 1 hour and it might not be straightforward to certain platforms, namely Windows), we provide pre-compiled executables for several platforms, that are automatically downloaded when executing the Clava framework. Currently Clava provides pre-compiled executables for Windows, Ubuntu, CentOS/Fedora and MacOS.

For Linux platforms, we provide a script, `clava-update`, which installs the tools available in the Clava framework to the folder where the script is. The Clava framework currently provides three tools, `clava` (the weaver), `clava-doc` (see Section 3.3) and `clava-unit` (see Section 3.4). Besides the tools, it also installs the CMake modules to the folder `/usr/local/lib/clava` (see Section 3.2). After installation, this script can be used to update the tools.

Finally, we have developed an online version of Clava with several examples, where users can immediately test the weaver [5].

3.2 CMake Integration

The build process of a C/C++ based project consists of several stages. First, the individual compilation units (.c or .cpp files) are compiled into binary object files. Then, the object files have to be linked together in the correct order, including third party libraries. After linking, the binary is packed into the requested format, which can either be a binary executable, a static library or a dynamic library. Each platform and operating system have multiple toolchains with various interfaces for building C/C++ code. Maintaining a complex project which targets multiple platforms in a hand-written build script (e.g., Makefile) can be inefficient and error-prone. There are tools which provide a higher-level of abstraction, by either working as generators for build scripts, or providing a platform- and system-independent implementation of this process.

CMake[6] is a popular tool for build management. Although it supports many languages, it is mainly used to compile C/C++ applications. Due its versatility, it quickly became quite popular in HPC environments. It can be used to generate standard Unix Makefiles, Visual

[5] `http://specs.fe.up.pt/tools/clava/`

[6] `https://cmake.org/`

Studio projects or configurations for other build systems. It features a highly flexible scripting language, which can be used to create modules for easier inclusion of third party tools. The modules can be used to amend the build process by introducing additional build steps or build targets to integrate custom code generators, static analysis tools or unit-testing frameworks.

Since Clava can be used for C/C++ source code manipulation and generation, CMake can be leveraged in this case by including an additional build step in which Clava is executed. We provide a CMake module that generalizes the application of LARA strategies on the source code tree and to allow multiple aspects to be applied on different parts of the source code. The module package, `ClavaConfig`, checks for Clava dependencies, downloads the Clava weaver executable if needed, and adds new functions to CMake (e.g., `clava_weave`). It can be also configured to use a local Clava instance by specifying a variable.

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 with the documentation from the source files. Since the LARA language is based on JavaScript, we adopted the JSDoc annotations[7], extended with LARA-specific annotations (e.g., `@aspect`, `@test`).

LaraDoc is part of the *Lara Framework*, and is independent from Clava. We added the utility *clava-doc*, which generates documentation taking into account the language specification of Clava. The Clava documentation for the Clava Standard Library (see Section 3.7) is generated using this tool and is available online (a link to documentation has been omitted from this document due to the blind review process).

3.4 Testing Framework

Clava includes a tool for performing unit testing. Figure 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 Figure 6).

3.5 Data Annotated in Source Code

Clava supports the attribute `getData`, which is used to access data that has been defined directly in the source code with the pragma `clava data`. Figure 8 shows an example where we use this pragma to annotate a function and define the key/value pairs value1/0 and value2/aString. When defining the key/value pairs we can use any code that is valid LARA or JavaScript.

After annotating the source code, this information can be accessed using the attribute `getData`, as shown in Figure 7.

3.6 Clava Actions

Clava provides built-in actions associated to certain join points, which can be called directly from LARA code. Below, we present some of the actions currently available in Clava, according to the join points to which they can be applied.

[7] `http://usejsdoc.org/`

```
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 will fail";

}
```

Figure 5: LARA test file with two unit tests.

```
LaraUnit test report

Failed tests:
 - FooTest.lara::TestFail (2.44s)
 [ERROR] User exception on line 21: This test will fail
 [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 21: This test will fail


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

Total Tests: 2
Passed / Failed: 1 / 1

 SOME TESTS FAILED
```

Figure 6: Example of a `clava-unit` test report.

```
#pragma clava data value1:0, value2:'aString'
void foo();
```

Figure 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.value1);
  println("Should be string aString: " + $function.value2);
end
```

Figure 8: Example of LARA code that uses the attribute `getData`.

3.6.1 Global Actions

Global actions are available for all join points, and include:

- `detach` - Removes the join point from the target code;
- `copy` - Creates a copy of the join point;
- `setValue` - Sets the value of an arbitrary property of the join point (e.g., line, filename);
- `messageToUser` - Sets a message to be printed to the user after weaving finishes. Identical messages are removed;
- `setUserField` - Associates arbitrary values in LARA to join points. These values can later be retrieved with `userField`;

3.6.2 `$program` Actions

- `rebuild` - Recompiles the current program, resolving literal code that might have been inserted. After recompilation, literal code becomes available for querying and manipulation;
- `push` - Creates a copy of the current program version and pushes it to the top of the program stack;
- `pop` - Discards the top-most program version of the program stack. Useful when used with textttpush for doing temporary modifications to the program that are to be discarded later. For instance, AutoPar-Clava (see Chapter 3) performs aggressive function inlining, in order to enable inter-procedural analysis. The changes done by the inlining are discarded after the analysis;
- External includes - Clava support several actions (e.g., `addExtraInclude`, `addExtraSourceFromGit`) to specify external sources (e.g., local files or folders, remote git repositories) that are not part of current program, but are necessary for its correct parsing or compilation. For instance, the Clava API that inserts code to measure energy (i.e., `lara.code.Energy`) requires a RAPL library, and uses this API to add information about where to find the source code for that library. Other strategies can use this information later, for instance, the API `lara.cmake.CMaker` that is used to compile the current version of the code;

3.6.3 `$file` Actions

- `rebuild` - Recompiles only a single file, instead of the entire program;
- `addGlobal` - Adds a global variables to the file;

- `write` - Writes the code represented by the file to a folder;
- `addFunction` - Creates a new function in the file with no parameters and no return, with a given name;

3.6.4 `$call` Actions

- `wrap` - Wraps the call around a function with a given name. The wrapping function is created if it is the first time the `wrap` action is used for the call of a given function;
- `inline` - Inlines the function call;

3.6.5 `$scope` Actions

- `addLocal` - Adds a new local variable to the scope;
- `clear` - Removes all instructions from the scope;

3.6.6 `$function` Actions

- `clone` - Creates a copy of the function, and renames and inserts the copy in the program (e.g., in the same file, in another file);
- `insertReturn` - Inserts code before all the return points of the function (i.e., return statements, and implicitly, the end of the function);
- `newCall` - Creates a new call to this function;

3.6.7 `$loop` Actions

- Canonical form loop setters - the join point `$loop` provides several setters related to canonical forms of C/C++ loops, such as for the initial and final value, condition, step and condition relation;
- `interchange` - Applies loop interchange to this loop and another given loop;
- `tile` - Applies loop tiling to this loop;

3.6.8 `$omp` Actions

The join point `$omp` provides several setters related to OpenMP pragmas. For instance, it is possible to set the values of the clauses `kind`, `numThreads`, `proc_bind`, `default`, `collapse`, `ordered` and `schedule`, and the variables that should appear in the clauses `private`, `firstprivate`, `lastprivate`, `shared`, `reduction` and `copyin`.

3.7 Clava Standard Library

One of the extensions that LARA provides over JavaScript is the `import` keyword, which enables basic modularity. We have used this import system to develop a standard library with APIs for the LARA framework, for Clava, and for the ANTAREX libraries[8].

[8] A comprehensive list of the APIs currently available in Clava can be found at `http://specs.fe.up.pt/tools/clava/doc/`

3.7.1 LARA Framework APIs

The LARA Framework APIs provide generic functionality that can be readily used by other weavers based on the LARA framework. This API includes:

- `lara.Io` - Basic I/O functionality related to files, such as read/copy/append/write/delete files or folders, write an object as a Json file, and calculate the MD5 of a file;
- `lara.Platforms` - Information about the current platform (e.g., if it is Linux, Windows or Mac);
- `lara.Strings` - Utility methods related to strings, such as escaping HTML and JSON strings, or generating a universal unique identifier (i.e., UUID);
- `lara.dse.DseLoop` - Performs Design-Space Exploration (DSE);
- `lara.cmake.CMaker` - Allows the creation of CMake configurations, for instance, to build the code for the current version of the target program from inside LARA. This functionality is useful for DSE loops that need to execute the code;
- `lara.code.*` - Package with classes for inserting specific kinds of code in the target program (e.g., logging, time and energy measurements);
- `lara.util.LineInserter` - Allows to modify the original code textually, instead of generating the code from the AST. This is useful in cases the original code needs to be preserved as much as possible;
- `lara.util.SequentialCombinations` - Generates sequences of combinations, according to the given elements. Used for exploring combinations of parameters;
- `lara.util.JpFilter` - Filters sets of join points, according to their attributes. Supports regular expressions;
- `lara.util.ProcessExecutor` - Launches command-line processes. Allows to set timeouts, and the possibility to retrieve the return value and separate strings for the standard output and error output of the executed application;
- `weaver.WeaverJps` - Contains functions to search join points, starting from the program or any arbitrary join point, using `lara.util.JpFilter` instances as filters. Allows to chain searches;

3.7.2 Clava APIs

Clava contains APIs specific to C/C++, such as:

- `clava.Clava` - Utility methods related to the execution of Clava, for instance, access to any option used to call Clava (e.g., user-specified compilation flags, the folder where the files transformed by Clava are written);
- `clava.ClavaJoinPoints` - Factory methods to create new join points (e.g., files, functions, statements, calls, variables, types, scopes and pragmas);
- `clava.autopar.*` - Package with auto-parallelization strategies for `for` loops, using OpenMP (see [14][15]);

- `clava.gprofer.Gprofer` - Profiles the current version of target code using the application `gprof`. Compiles and runs the current version of the target program, and provides methods to access the information obtained by `gprof` after running the application;
- `clava.hdf5.Hdf5` - Automatically generates HDF5 wrappers for C++ structures and classes (see [16]);
- `clava.mpi.MpiScatterGatherLoop` - Applies an MPI scatter-gather strategy to loops;
- `clava.opencl.KernelReplacer` - Replaces a function call with an equivalent call to a user-provided OpenCL kernel. Takes care of inserting all the required boiler-place code in the host code to call the OpenCL kernel;
- `clava.util.SingleFile` - Merges all current files in a single output file. Useful for certain compilation toolchains;

3.7.3 ANTAREX APIs

We provide APIs for each one of the technologies developed during the ANTAREX project. These APIs include:

- `antarex.examon.Examon` (see Chapter 9) - Adapts the current target program to use the Examon library for highly scalable performance and energy monitoring of HPC servers;
- `antarex.libvc.LibVC` (see Chapter 4) - Enables runtime compilation of source code and dynamic loading of a specified C/C++ function (see [17]). It also provides support for versioning of the compiled functions;
- `antarex.margot.*` (see Chapter 8) - Enables dynamic adaptation of applications, in order to face changes in the execution environment or in the application requirements (see [18]);
- `antarex.memoi.*` (see Chapter 6) - Enables memoization of calls to arbitrary pure functions;
- `antarex.precision.CustomPrecision` (see Chapter 5) - Allows to change the types/-precision of variables in the code;
- `antarex.split.*` (see Chapter 4) - Enables split compilation in a target application;

3.8 Miscellaneous Features

Clava provides the following miscellaneous features:

- *Parallel Parsing* - Clava supports parsing the source files in parallel. This allows to considerably speedup parsing time for projects that contains large numbers of source files;
- *Mixed C/C++ and OpenCL projects* - it is possible to specify in the same Clava compilation project C/C++ and OpenCL files. In this case, both types of files become part of the AST, and it is possible to write strategies that require information about the two kinds of files (e.g., change the parameters of an OpenCL kernel and at the same type adapting its call in the C/C++ host code);

```
 1 import lat.Lat; // Import the 'Lat' class
 2
 3 aspectdef LatExample
 4   var lat = new Lat("myFirstLat"); // Create the LAT object
 5
 6   var xValues = new LatVarList("x", [10, 20, 30]); // Values for variable x: 10, 20, 30
 7   var yValues = new LatVarRange("y", 1, 5);  // values for variable y: 1, 2, 3, 4
 8
 9   lat.addSearchGroup([x, y]); // Combine the values of x and y
10
11   // Select the loops in the source code
12   select loop end
13   apply
14     lat.setScope($loop); // The region of code where the values will be set
15     break; // Measure just the first loop that was found
16   end
17
18   lat.tune(); // Test the variants and create the report
19 end
```

Figure 9: LARA code that uses the LAT tool.

- *Partial Parsing* - Clava supports partially parsing code that has syntax errors as an option. Statements that have errors are removed from the final parsed AST;

4 Clava Use Cases

This section presents a sample of the work that has been done by people that have used Clava to write LARA strategies, as an example of the range and current capabilities of the tool. Most of the work presented in this section have been already published.

4.1 LAT - Lara Auto Tuner

LAT is an implementation of the Intel Software Autotuning Tool (ISAT) [19] built entirely on LARA. It can compile, run and test multiple values for source-code variables, and provides a report about what are the best variants.

Figure 9 shows a LARA strategy that uses the LAT tool. First, imports the LAT tool (line 1) and creates an instance of the LAT class (line 4). Then, defines which values should be explored, for which variables in the source code. In this case, the strategy creates a set of discrete values for the variable `x` (line 6), an a range of values for variable `y` (line 7), and adds them to the current tuning operation (line 9).

Lines 12-16 select the loops in the current C/C++ source code, and set the first loop that is found as the scope where the values of `x` and `y` are changed (lines 14-15). When setting the scope, implicitly the measurements will be done around that same scope. By default, LAT measures execution time, but it allows to set other metrics (e.g., energy consumption).

Finally, line 18 executes the autotuning. This generates one version of the target C/C++ code for each combination of values for the variables `x` and `y`, compiles them, runs them, and extracts the metrics. In the end, it generates a report with the results for all the variants of the code.

```
 1 import clava.autopar.Parallelize;
 2
 3 aspectdef AutoPar
 4
 5   var $loops = [];
 6   select function{"foo"}.loop end
 7   apply
 8     $loops.push($loop);
 9   end
10
11   Parallelize.forLoops($loops);
12 end
```

Figure 10: LARA code that uses the AutoPar library.

4.2 AutoPar - OpenMP Parallelization

AutoPar-Clava is a Clava library written in LARA for automatic parallelization of `for` loops in C code, with previous versions presented in [15] and [14], and with the current version described in Chapter 3. It performs static analysis of data dependencies between iterations of loops, to determine if it is possible to parallelize a loop. If AutoPar concludes that a loop can be parallelized, it then generates a OpenMP pragma that is inserted before the loop.

Figure 10 shows a LARA strategy that uses the AutoPar library. First, imports the utility class Parallelize (line 1), available in AutoPar. Then, selects all the loops inside the function with name `foo` and stores them in the array `$loops` (lines 5-9). Finally, calls the function `Parallelize.forLoops()` (line 11), which attempts to parallelize all the given loops. Loops that could be parallelized will have an OpenMP pragma before them after the function `forLoops` finishes. Loops that could not be parallelized remain unchanged, and a warning message is given to the user explaining why each particular loop could not be parallelized.

4.3 OpenCL Integration

The package `clava.opencl` provides classes related with integration of OpenCL kernels in C/C++ code. OpenCL is a programming language for writing kernels of computation that should be accelerated. One of its main features is that the same OpenCL kernel can run on many different kinds of systems (e.g., CPUs, GPUs), making it an interesting option for heterogeneous platforms. When adapting a C/C++ to use OpenCL, the portion of the program where most of the computation is done (i.e., the hotspot) is rewritten as an OpenCL kernel. The rest of the program is then adapted to interface with the OpenCL kernel. However, interfacing with a single OpenCL kernel usually requires dozens of lines of C/C++ code.

The class `KernelReplacer` is a Clava library that replaces a call to a target function with a call to an equivalent, user-provided, OpenCL kernel. This allows the user to quickly change the implementation of one of the hotspots of the application with a more efficient version that can run on a GPU or other accelerators. The user needs to provide the original application, the intended OpenCL kernel in a separate file, and the OpenCL configuration for the call. From this information, the library generates all the needed boilerplate code to interact with the OpenCL platform, from choosing the correct device to generating and managing the buffers and their memory transfers.

Figure 11 shows a LARA strategy that uses the class `KernelReplacer`. First, the class is imported (line 1). Then, defines several parameters, such as the location of the OpenCL kernel file (line 6) or the sizes of the kernel arrays (lines 8-12). The next step is to choose the function

```
 1 import clava.opencl.KernelReplacer;
 2 
 3 aspectdef KernelReplacerExample
 4 
 5   // path relative to the file where the target call is
 6   var kernelCodePath = '../cl/gemm.cl';
 7 
 8   var bufferSizes = {
 9     A: "N*M*sizeof(double)",
10     B: "M*K*sizeof(double)",
11     C: "N*K*sizeof(double)"
12   };
13 
14   select stmt.call{'matrix_mult'} end
15   apply
16     var kernel = new KernelReplacer($call,
17       "mat_mul_kernel", kernelCodePath,
18       bufferSizes,
19       [1, 64], ['N', 'K']);
20 
21     kernel.setOutput('C');
22 
23     kernel.replaceCall();
24   end
25 end
```

Figure 11: LARA code that uses the KernelReplacer class.

call that will be replaced by a call to the OpenCL kernel (line 14) and create an instance of `KernelReplacer` with all the needed information (lines 16-21). Finally, replaces the original function call with all the OpenCL code needed to enqueue and execute the kernel (line 23).

4.4 Memoization

The package `memoi` provides classes that can apply the Memoization technique to C/C++ code. Memoization is an optimization technique that caches results of expensive computations. This is performed on pure functions, since their output depends only on their input. Whenever the computation is required again for an input that was already cached, we can retrieve the stored result instead of executing the called function again. This LARA library is capable of applying the memoization technique to function calls in the source code, replacing them with calls to a memoization library. The package `memoi` has been developed by INRIA and is described in Chapter 6.

Figure 12 shows a LARA strategy that uses the class `Memoization`. First, imports the class `Memoization`, which contains several memoization aspects (line 1). The first aspect to be called is an initialization stage (line 4). Then, identifies the mathematical functions from the system header to be memoized (line 5), as well as user functions (line 6). Finally, calls a finalization aspect (line 7). After this the target C/C++ source code is enhanced with memoization.

5 Related Work

A number of approaches allow the specification of non-functional concerns (e.g., code transformations, compiler optimizations) in DSLs that are then applied over a given target code.

CHiLL [20] is a declarative language focused on strategies for loop transformations. CHiLL strategies are scripts, written in separate files, which contain a sequence of transformations to

```
import antarex.memoi.Memoization;

aspectdef Launcher
  call Memoize_Initialize( );
  call Memoize_MathFunctions(['cos' , 'acos' , 'sqrt']);
  call Memoize_Function('myfunc');
  call Memoize_Finalize( );
end
```

Figure 12: LARA code that uses the Memoization class.

be applied in the code during a compilation step. The PATUS framework [21] defines a DSL specifically geared toward stencil computations and allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations (e.g., loop unrolling) and architecture-specific extensions (e.g., SSE). This is similar to the way Clava (and LARA) supports defining specific actions for individual join points (e.g., `$loop.tile`, `$loop.interchange`).

Term-rewriting is another approach for code analysis and transformation, as is the example of Stratego/XT [22], or Rascal [23]. In the context of code transformations, term rewriting is based on representing the source code as `terms`, a symbolic representation that is then manipulated by rules. In this approach, strategies tend to be more declarative than imperative, the user defines rewrite rules and the term rewriting framework decides when and where the rules will be applied. In one hand, this can simplify the strategies, on the other hand it can also limit the applicability range of the approach.

Aspect-Oriented Programming (AOP) approaches inject functionality and change the runtime behavior of an application, according to rules defined in *aspects* that are written in a DSL, in files outside of the target source code. Most well-known AOP approaches use as DSL an extension of their target language. For instance, AspectJ [24] is an AOP DSL that extends Java with constructs that allow to select points in the code (e.g., `call(set*(..))` selects all calls whose name have the prefix `set`). AspectC++ [25] is an AOP extension to the C++ programming language inspired by AspectJ, and uses similar concepts, adapted to C++. Code selection in AOPs is usually restricted to object-oriented lexical constructors, such as classes, method calls and fields. For instance, unlike Clava, both AspectJ and AspectC++ are unable to select local variables, statements, loops, and conditional constructs.

6 Conclusion

This chapter 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. We presented the framework, its structure and the principles behind its design, and a number of simple, but representative, examples that show some of the features of the Clava + LARA approach.

The Clava framework provides an aspect-oriented programming approach, implemented by an internal weaver and the technology provided by the LARA DSL, in order to describe source-to-source strategies, such as code transformations and instrumentation. Clava users can program their own strategies and take advantage of a set of libraries already available.

7 Clava Tutorial

In this hands-on session we introduce the LARA DSL and the Clava source-to-source compiler, and use these technologies to apply several strategies for code analysis and transformation.

7.1 Getting Started

7.1.1 Linux Preparation

1. Get the tutorial files from `http://specs.fe.up.pt/tutorials/2018-ANTAREX-Book.zip`
2. Unzip the tutorial files (the new directory is called `<BASE>`)
3. Get the script from `http://specs.fe.up.pt/tools/clava/clava-update`
4. Put it in a place on the path
5. Run it in a terminal (may need chmod and sudo)
6. Start Clava by running the command `clava`

7.1.2 Windows Preparation

1. Get the tutorial files from `http://specs.fe.up.pt/tutorials/2018-ANTAREX-Book.zip`
2. Unzip the tutorial files (the new directory is called `<BASE>`)
3. Get the jar from `http://specs.fe.up.pt/tools/clava.jar`
4. Get The CMake files from `https://bit.ly/2EVVnF7`
5. Unzip the files and leave them in an accessible place
6. Start Clava by running the command `java -jar clava.jar`

This runs Clava in GUI mode. In the first tab, *Program*, you can see a field to load a configuration file, a button to start the execution and an output area. In the second tab, *Options*, you can set the options for a specific configuration, or create a configuration file that can be loaded in the *Program* tab. Finally, the third tab, *LARA Editor*, allows you to edit your LARA strategies and target source code.

7.2 Call Graph

7.2.1 Main Idea

This first section introduces the basics of the LARA language. This example also shows how arbitrary JavaScript can be used in a LARA strategy.

The idea is to build a static call graph based on the target source code. Each node will represent a function, and an edge from node A to node B means that function A calls function B. The weight of the edge will be the number of function calls that can appear in the source code.

To build this graph, we select tuples of in the form $< function, call >$, and increment a counter each time a tuple is found. At the end it outputs the graph in *dot* format.

7.2.2 Implementation

First, we start by loading the configuration file for the first example. In the tab *Program*, click *Browse...* and open the file:

```
<BASE>/1.CallGraph/CallGraph.config
```

Click the tab *Options* to check the values that were loaded. The field *Aspect* is the LARA strategy to be applied to the target code, the field *Sources* define the source files of the target code, and the field *C/C++ Standard* sets the language standard to be used. Right now you do not need to worry about the remaining options.

Click the tab *LARA Editor* and you can see the LARA aspect to be applied. The code starts with the `aspectdef` keyword, which defines a strategy.

The `select` block in line 7 is used to select all functions in the code and then all calls from within each of the selected functions. In the `apply` block in line 8, the weaver will iterate over each of these pairs. We use JavaScript code to keep a count of each pair we have seen.

To execute the LARA strategy, you can either go to the tab *Program* and click the button *Start*, or stay in the *LARA Editor* tab and either click the play button, or press `F11`.

After execution, the output area should show a graph in *dot* format. To see the graph, copy the code of the graph, open the web page `http://webgraphviz.com/`, paste the code and click *Generate Graph!*.

7.3 Logging

7.3.1 Main Idea

This example instruments code in order to log fine-grained application events. More specifically, it prints a message each time we are about to execute a loop.

7.3.2 Inserting C Code

Open the following configuration file:

```
<BASE>/2.Logging/1.Inserts.config
```

Select the editor tab and check the LARA code. The strategy selects loops and inserts code for printing before each loop. Besides the loops, we also select the corresponding `file` and `function`, in order to work with them in body of the *apply*.

Click the play button or press `F11` to apply the LARA strategy. The output window show the modified code, and the loop should now have a `printf` call right before it.

7.3.3 Exercise: Inserting C++ Code

Open the following configuration file:

```
<BASE>/2.Logging/2.InsertsExercise.config
```

The LARA strategy is incomplete! In this exercise we suggest you to complete the strategy so that it inserts code equivalent to the previous exercise, but using idiomatic C++ code (e.g., `std::cout`*). Do not forget to add the correct headers..*

7.3.4 Using the Logger API

Direct insertion of code as used in the previous exercise is very flexible, but can be cumbersome and error-prone. To alleviate this, LARA supports the development of libraries and APIs that provide a higher level of abstraction.

Open the following configuration file:

```
<BASE>/2.Logging/3.Api.config
```

Instead of direct insertions of source code, this strategy uses a logger library. To use the logger library, first we import it (line 1), and then we instantiate the Logger (line 5). Then, we use the functions available in the Logger object to build the text we want to print. Apply the strategy, and the code in the output window should have C++ printing-related code before the loop.

The same LARA library can have different implementations according to the target language. Open the following configuration file:

```
<BASE>/2.Logging/4.Api-C.config
```

This configuration changes the input code, which now is C, and the compilation standard in the configuration (C11 instead of C++11). The LARA strategy is the same as in the previous example. Apply the strategy, and the code that now appears in the output window should have printing code before the loop that uses `printf` instead of `std::cout`.

7.4 Measurements

7.4.1 Main Idea

We can use LARA to collect different metrics in several parts of the application, controlled by the selection and filtering of the points of interest. In this example, we change the application to measure execution time and energy consumption around loops.

While we can use direct insertions to add logging and measuring code, we recommend using APIs whenever possible. We provide reference documentation in the link `http://specs.fe.up.pt/tools/clava/doc/`. There you can find the documentation for the `Logger` API presented in the previous example, as well as for all APIs supported natively by Clava.

7.4.2 Measuring Time

Open the following configuration file:

```
<BASE>/3.Measurements/1.Time.config
```

The LARA strategy imports a new library, Timer (line 1). It is then instantiated (line 6) and used to insert code for time measurement (line 10). The call to `time` inserts measuring code around the provided code point (a loop in this case) and prints the result. Apply the strategy, the code in the output window should have C-specific code around the loop.

Open the following configuration file:

```
<BASE>/3.Measurements/2.Time-CPP.config
```

This configuration uses the same LARA strategy, but changes the configuration to interpret the target source code as a C++. Apply the strategy and check the code in output window, the measuring and printing code inserted around the loop should be C++.

7.4.3 Exercise: Measure Energy Consumption

Open the following configuration file:

```
<BASE>/3.Measurements/3.EnergyExercise.config
```

The LARA strategy is incomplete! In this exercise we suggest you to use Clava APIs to measure both time and energy around loops. Use the `lara.code.Energy` API and the reference documentation (`http://specs.fe.up.pt/tools/clava/doc/`).

7.5 AutoPar

7.5.1 Main Idea

This example uses the AutoPar library to analyze and parallelize `for` loops, and introduces the Clava CMake plugin.

7.5.2 Auto-parallelization With Clava

Using the terminal, go to the following folder:

```
cd <BASE>/4.AutoPar/
```

This folder has the application code in the subfolder `src` and the LARA strategy in the subfolder `lara`. The application is a simple matrix multiplication that has a pragma marking the outer loop of the multiplication kernel.

Check the LARA code using the following command:

```
gedit lara/AutoPar.lara &
```

The LARA strategy imports the `AutoPar` library (line 1), selects the loop marked with a pragma (line 6) and parallelizes it.

Open the `CMakeLists.txt` file:

```
gedit CMakeLists.txt &
```

This is a regular CMake build script, with the exception of the last two lines, which use the Clava CMake plugin. Line 18 imports the plugin, and line 20 applies the LARA strategy `AutoPar.lara` in the CMake target `matrix_mul`. The CMake function `clava_weave` is equivalent to applying the LARA strategy using the GUI.

Build the application using the standard CMake steps:

```
mkdir build
cd build
cmake ..
```

The building process searches for Clava in your system and applies the LARA strategy over the application before the compilation and linking stages. The strategy prints the modified code. Please check the output and verify that the code now has OpenMP pragmas.

Finish building the application:

```
make
```

7.6 Exploration

7.6.1 Main Idea

This example uses `LAT`, a Clava third party library that performs design-space exploration over values of code variables. We use LAT to explore the impact of the number of threads after parallelizing an application with `AutoPar`.

7.6.2 Exercise: Apply Exploration after Auto-Parallelization

Using the terminal, go to the following folder:

```
cd <BASE>/5.Exploration/
```

In this folder, the application code is in the subfolder `src` and the LARA strategy in the subfolder `lara`. The application is the same as in the previous section.

Check the LARA code using the following command:

```
gedit lara/Exploration.lara &
```

The LARA strategy creates and setups a LAT object that performs design-space exploration on the number of threads.

Open the `CMakeLists.txt` file:

```
gedit CMakeLists.txt &
```

The CMakeLists.txt file is incomplete! In this exercise, we suggest you to use the Clava CMake plugin to first parallelize the code and then explore the number of threads.

If you tried to build the application and if that failed, do not worry, this was supposed to happen. Your code should be similar to the following:

```
clava_weave(matrix_mul lara/AutoPar.lara)
clava_weave(matrix_mul lara/Exploration.lara)
```

Since the `LAT` library is a third-party library, it is not distributed with Clava, so we need to include it, by either specifying a path to a local folder, or an URL to a git repository. The function `clava_weave` supports passing flags to Clava, and we can use this mechanism to tell Clava where to find the `LAT` library.

Modify the call to the strategy `clava_weave` in the following way:

```
clava_weave(matrix_mul lara/Exploration.lara
    FLAGS
    -dep
    https://github.com/specs-feup/LAT-Lara-Autotuning-Tool.git)
```

Build the application using the standard CMake flow:

```
mkdir build
cd build
cmake ..
```

Inside the `build` folder there should now be a subfolder called `dse`. This folder contains all the versions that were generated and tested, as well as the results of the exploration.

Open the `LAT` report and check the results of the design-space exploration:

```
firefox dse/results/report_dse_0.html &
```

7.7 mARGOt Integration

7.7.1 Main Idea

In this example we show how we can use Clava to integrate the mARGOt autotuner into an application. This example takes care of building the configuration file and the knowledge base to start the autotuning process. The last phase changes the code of the target application to include calls to the mARGOt API.

7.7.2 The Top Level Strategy

Open the following configuration file:

```
<BASE>/6.mARGOt/mARGOt.config
```

Select the editor tab and check the LARA code. This is the top-level aspect that controls the three main steps of the strategy for the integration of mARGOt. First, the XML configuration is generated with the call to `XmlConfig`. In this aspect one can see the LARA library for mARGOt configurations. Then, the call to the aspect `Dse` performs the design-space exploration needed to build the initial knowledge base of the autotuner. This is performed using the LARA library for mARGOt exploration, which is implemented using the `LAT` library. Finally, the call to the aspect `CogeGen` inserts calls to the mARGOt API in specific points of the target application. This API is generated at build time based on the XML configuration generated in the first step. The code generation and insertion is performed with the LARA library for mARGOt code generation.

At this point, one can decide to weave the code or examine each of the sub-aspects that make this strategy. To weave one can click the play button or press `F11`.

7.7.3 Building the Final Application

To build the application with mARGOt support, one can follow these steps:

```
cd <BASE>/6.mARGOt/
cp scripts/* woven/
cd woven
./build_scenario.sh
```

This downloads the mARGOt code, builds the main library and the interface (according to the XML configuration), and compiles the target application linked to the mARGOt library. To run the built application, one can do:

```
build/mm
```

7.8 Clava Actions

7.8.1 Main Idea

In this example, we show how to use Clava actions, namely loop transformations. Actions act over a selected join point and may be parameterized. Furthermore, they are called within an apply block, which means one can perform any allowed filtering to the join points and target only the ones one are looking for.

7.8.2 Loop Transformations

Open the following configuration file:

```
<BASE>/7.Transformations/Tile.config
```

If you go to the editor tab you can see an example aspect that performs loop tiling on a selected loop, filtered by the control variable. The `tile` action is invoked with the `exec` keyword. In this example, we specify a tile size of 64 and the inclusion of the new loop (which iterates over each tile) immediately before the target loop.

One can click the play button or press `F11` button to weave this aspect into the application, and check the result in the file:

```
<BASE>/7.Transformations/woven/output/src/matrix_mul.c
```

Now load the following configuration file:

```
<BASE>/7.Transformations/Interchange.config
```

In the editor tab there is an example aspect that performs loop interchange and uses some more features of LARA. In that example, two loops are selected, one being the *parent* of the other. We assign them variable names, so we can later use them in the `apply` and `condition` blocks. We `exec` the loop interchange action on one of the loops and pass the other as an argument (the order could be reversed). In the `condition` block we filter both loops using their control variables.

7.9 Multiversioning

7.9.1 Main Idea

With this example we illustrate a complex use of Clava and the LARA language. In this example, Clava generates a different version of a program to work with a different data type. The user selects a function call to be replaced with a switch that chooses between the original version and the newly generated one. The switch is controlled by a user defined variable. The function definition of the selected call and all functions below it (on the subgraph of the call graph that has that function as root) are cloned. In every cloned function, parameters, local variables and return type are changed from double to float.

7.9.2 The Top Level Strategy

Open the following configuration file:

```
<BASE>/8.Multiversion/Multiversion.config
```

One can see the top level strategy in the editor tab. Since parts of this strategy are complex on their own and may be reused, we organized them into different aspects and LARA files (these are imported at the top of the main file, `Multiversion.lara`).

The main aspect selects a function call and adds a local variable (to control the switch) on the same scope, then it calls the `Multiversion` aspect. In here we replace the statement encapsulating the original call with a switch. This is performed in the `CreateSwitch` aspect. The condition of the switch is the local variable included before and each of the cases is a call to one of the versions.

At the end of this aspect, another aspect is called, once for each function call in the switch. The aspect `MeasureTimeAndEnergy` instruments each of the calls in order to measure energy consumption and execution time of that particular version.

The bulk of the work for this strategy is performed in the `CreateFloatVersion` aspect. One can open it from the left side panel, where all the aspect files are listed. Two aspects are called from this. First, `CloneFunction` is called, which recursively clones every function in the subgraph of the call graph that has our target function as root. Following this, Clava iterates through every clone and call `ChangePrecision` on each of the clones. This aspect changes the type of the parameters, local variables and return type.

One can weave this strategy into the application by clicking the play button or pressing the `F11` button.

8 Acknowledgments

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] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[2] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[3] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[4] David H Jones, Adam Powell, Christos-Savvas Bouganis, and Peter YK Cheung. Gpu versus fpga for high productivity computing. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 119–124. IEEE, 2010.

[5] Lois Goldthwaite. Technical report on c++ performance. *ISO/IEC PDTR*, 18015, 2006.

[6] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[7] Chuck Pheatt. Intel&reg; threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[8] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[9] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[10] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[11] João MP Cardoso, José GF Coutinho, Tiago Carvalho, Pedro C Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 46(2):251–287, 2016.

[12] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es):154, 1996.

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

[14] Jorge Barbosa João MP Cardoso Hamid Arabnejad, João Bispo. An openmp based parallelization compiler for c applications. In *16th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2018)*. IEEE, 11-13 Dec., 2018.

[15] Hamid Arabnejad, João Bispo, Jorge G. Barbosa, and João M.P. Cardoso. Autopar-clava: An automatic parallelization source-to-source tool for c code applications. 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*, PARMA-DITAM '18, pages 13–19, New York, NY, USA, 2018. ACM.

[16] Martin Golasowski, João Bispo, Jan Martinovič, Kateřina Slaninová, and João M. P. Cardoso. Expressing and applying c++ code transformations for the hdf5 api through a dsl. In Khalid Saeed, Władysław Homenda, and Rituparna Chaki, editors, *Computer Information Systems and Industrial Management*, pages 303–314, Cham, 2017. Springer International Publishing.

[17] S. Cherubin and G. Agosta. libversioningcompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions. *SoftwareX*, 7:95 – 100, 2018.

[18] D. Gadioli, E. Vitali, G. Palermo, and C. Silvano. margot: a dynamic autotuning framework for self-aware approximate computing. *IEEE Transactions on Computers*, pages 1–1, 2018.

[19] Intel software autotuning toolintel software autotuning tool, Retrieved: 15-09-2018. `https://software.intel.com/en-us/articles/intel-software-autotuning-tool`.

[20] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 136–150. Springer, 2010.

[21] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 676–687. IEEE, 2011.

[22] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52 – 70, 2008.

[23] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 168–177. IEEE, 2009.

[24] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer, 2001.

[25] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. 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, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.