H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

http://www.antarex-project.eu/

Deliverable D2.2: Code Refactoring Tool Guided by DSL

European Commission

Horizon 2020
European Union funding
for Research & Innovation

Deliverable Title:	Code Refactoring Tool Guided by DSL		
Lead beneficiary:	UPORTO (Portugal)		
Keywords:	Clava, LARA DSL, prototype, accompany report		
Author(s):	João M.P. Cardoso (UPORTO); João Bispo (UPORTO); Pedro Pinto (UPORTO); Tiago Carvalho (UPORTO);		
Reviewer(s):	Giampaolo Agosta (POLIMI); Stefano Cherubin (POLIMI); Imane Lasri (INRIA);		
WP:	WP2	**Task:**	T2.2
Nature:	Other	**Dissemination level:**	Public
Identifier:	D2.2	**Version:**	V3.3
Delivery due date:	August 30th, 2017	**Actual submission date:**	Sept. 8th, 2017

Executive Summary: This document is the accompanying report associated with the **Deliverable D2.2** released as the **Clava: C++ language + LAra weaver And code transformer**. The Clava tool represents the results of the activities carried out by the project partners from M09 to M24 in **Task 2.2 "DSL Front-end Compiler and Interpreter for Adaptivity"** under the leadership of UPORTO.
This deliverable is organized as follows:

- Section 1 introduces the deliverable;
- Section 2 is about the LARA frontend components;
- Section 3 introduces the Clava source to source C/C++ compiler;
- Section 4 concludes the deliverable.

Clava is released open-source under the Apache 2.0 license, publicly available at https://github.com/specs-feup/clava

Approved and issued by the Project Coordinator: [illegible]	**Date:** **Sept. 8th, 2017**

Project Coordinator: Prof. Dr. Cristina SILVANO – Politecnico di Milano
e-mail: silvano@elet.polimi.it - **Phone:** +39-02-2399-3692- **Fax:** +39-02-2399-3411

Table of Contents

1 Introduction 4
2 LARA Frontend Components 5
2.1 LARA Tool (larac) 5
2.2 LARA Interpreter (larai) 5
3 About Clava 7
3.1 Software Architecture 7
3.2 Join Point and Attribute Models 8
3.3 How to Install Clava 8
3.4 How to Run Clava 8
3.4.1 GUI Version 9
3.4.2 Command Line Version 10
3.5 Examples 11
4 Conclusions 12
5 References 13
6 Acronyms and Abbreviations 14

1 Introduction

This deliverable provides the DSL front-end enhanced to process a LARA version with extensions for runtime adaptivity. `Clava` is a source-to-source compiler responsible to transform C/C++ code according to the input strategies specified in LARA. The framework proposed consists of the `Clava` tool, augmented with libraries for runtime adaptivity strategies integrated in the ANTAREX compilation flow.

All the tools of the framework presented in this deliverable can be downloaded from:

```
http:/www.fe.up.pt/~specs/projects/lara/doku.php?id=lara:downloads
```

And an online version of `Clava` is available at:

```
http://specs.fe.up.pt/tools/clava/
```

Documentation about the tools, about the LARA DSL, and examples are provided at the LARA wiki:

```
http:/www.fe.up.pt/~specs/projects/lara/doku.php?id=lara:documentation
```

The minimum execution requirements to execute `Clava` is the installation of Java 8 runtime[1] [5] in the operating system. We have tested `Clava` in Windows 10, Ubuntu 14.04 and 16.04, and CentOS 6.7.

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html

2 LARA Frontend Components

The LARA DSL is supported by two main components:

- The LARA compiler, `larac`, which is responsible to process the input LARA code and to translate it to an intermediate representation, Aspect-IR.
- The LARA interpreter, `larai`, which is responsible to interpret LARA code (using as input the Aspect-IR representation).

Both larac and larai can be downloaded from the LARA wiki [1] (see webpage: http:/www.fe.up.pt/~specs/projects/lara/doku.php?id=lara:downloads). They consist of two jar files: larac.jar and larai.jar. We note that larai can be used in a standalone mode as larac is included in the larai.jar file available. The two tools are integrated in the weavers and executed without user intervention when a compiler including a weaver is executed. This is the case with the `Clava` source to source compiler as described in Section 3: About .

2.1 *LARA Tool (larac)*

The `larac` tool processes a LARA input file and generates the Aspect-IR represented in an XML (see Figure 1). This tool includes a lexical, a syntactic and a semantic analyser.

`Larac` can be downloaded from:

```
    http://specs.fe.up.pt/tools/larac.jar
```

`larac` can be executed in the following way:

```
    java -jar larac.jar myaspect.lara
```

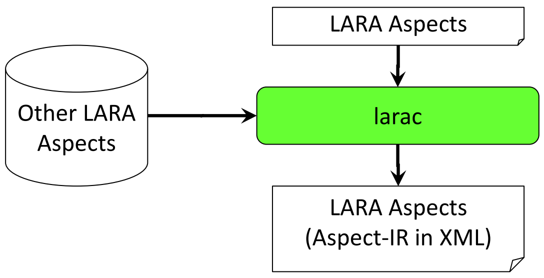


Figure 1. The `larac` flow.

`Larac` includes the join point, attribute and action models as internal structures. These models fundamentally depend on the input programming language and on the possibilities to execute external tools in the case of using `larai` in standalone mode (see the following section).

2.2 *LARA Interpreter (larai)*

The `larai` tool is able to execute LARA or Aspect-IR files which do not use source-to-source features. For instance, execution of the code "`select file end`" is not supported by `larai` itself, but added on top of `larai` by tools such as `Clava`. Figure 2 shows the flow of `larai` (which includes and uses `larac`). `larai` also supports the definition of interfaces to external tools (e.g., gcc, llvm) which can be called seamlessly from inside LARA aspects.

`larai` can be downloaded from:

```
    http://specs.fe.up.pt/tools/larai.jar
```

`larai` can be executed in the following way:

```
    java –jar larai.jar myaspect.lara
```

or

```
    java –jar larai.jar myaspect.ir
```

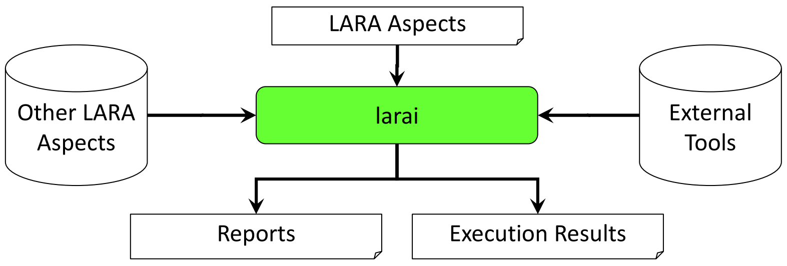


Figure 2. The `larai` flow.

3 About Clava

`Clava` is a source to source compiler guided by LARA and fully developed during ANTAREX. The compiler receives as input C/C++ and LARA code and produces C/C++ code (see Figure 3). The main objectives of `Clava` are the following:

- To provide automatic insertion of instrumentation, monitoring, and logging code in application code;
- To provide support for code refactoring, code transformations, and split-compilation, guided by LARA strategies;
- To provide the necessary interface and monitoring for runtime autotuning;
- To provide code modifications according to parallelization strategies (e.g., targeting OpenMP) defined in LARA;

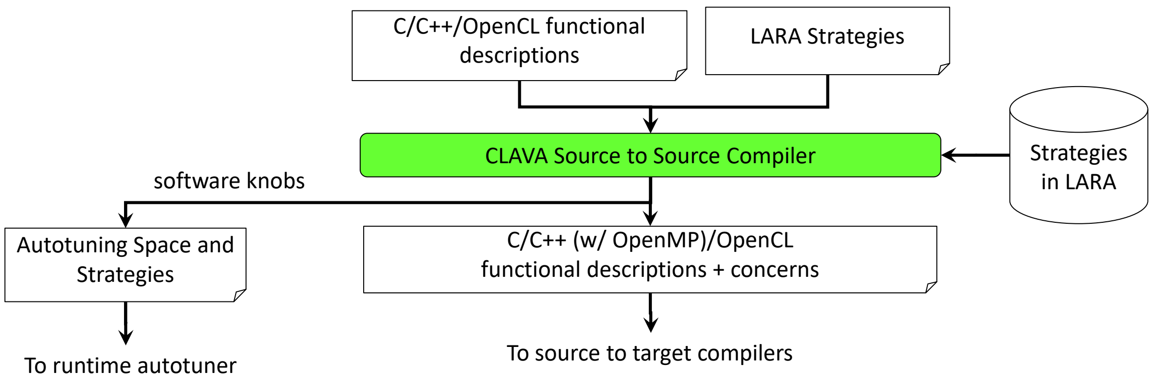

C/C++/OpenCL functional descriptions
LARA Strategies
CLAVA Source to Source Compiler
Strategies in LARA
software knobs
Autotuning Space and Strategies
C/C++ (w/ OpenMP)/OpenCL functional descriptions + concerns
To runtime autotuner
To source to target compilers

Figure 3. `Clava` input/outputs[2].

3.1 *Software Architecture*

Figure 4 shows the `Clava` software architecture. `Clava` integrates a `Clang` [2] based front-end which is responsible to parse C/C++ files and to output an abstract syntax tree (AST). This AST and the LARA code are then processed by the weaver integrated in `Clava`. As shown in the figure, `larac` and `larai` process the input LARA code and interface with the weaver in order to command actions and request attribute values.

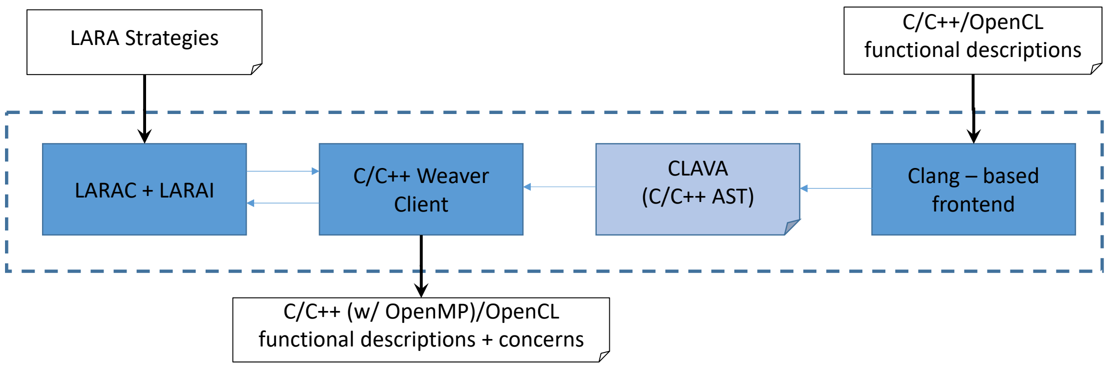

LARA Strategies
C/C++/OpenCL functional descriptions
LARAC + LARAI
C/C++ Weaver Client
CLAVA (C/C++ AST)
Clang – based frontend
C/C++ (w/ OpenMP)/OpenCL functional descriptions + concerns

Figure 4. Architecture of `Clava`.

[2] At the moment Clava is not supporting OpenCL as input code, but we have plans to extend it in order to deal with OpenCL code.

3.2 Join Point and Attribute Models

The join point model used by Clava [3] includes an extensive set of points of interest in C/C++ code, such as method invocation, variable declarations, loops, conditional constructs, and structs. The attribute model [3] also includes many of the properties of the join points statically provided by the weaver.

3.3 How to Install Clava

Clava is a compiler written in Java which has been developed using Eclipse and depends on several other projects and libraries. To build Clava without setting up the project in Eclipse, one can download the program EclipseBuild:

```
    http://specs.fe.up.pt/tools/eclipse-build.jar
```

And run the following command:

```
    java    -jar    eclipse-build.jar    --config    https://raw.githubusercontent.com/specs-feup/clava/master/ClavaWeaver/eclipse.build
```

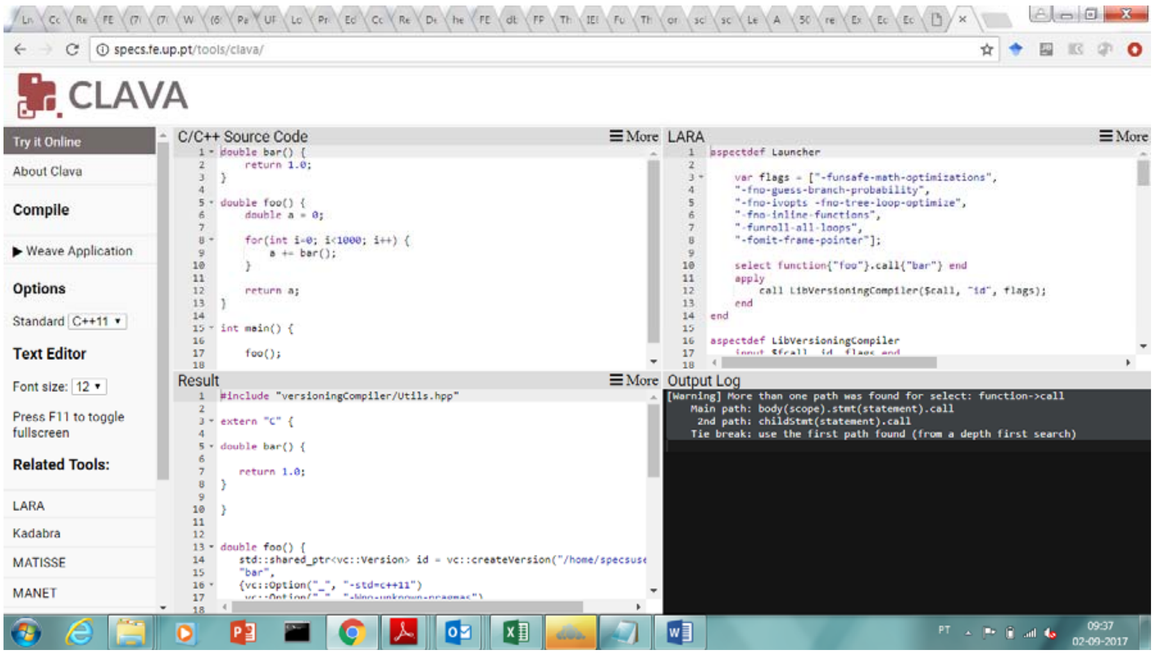
This will generate the file ClavaWeaver.jar. Clava can also be downloaded using the links provided in the LARA wiki [1] or directly the following link:

```
    http://specs.fe.up.pt/tools/clava.jar
```

In order to run Clava the host system needs to have the Java runtime (JRE) [5] installed, version 8 or superior. In order to compile Clava the host system needs to have the Java Development Kit (JDK) [5] installed, version 8 or superior.

3.4 How to Run Clava

Clava can be used as an online compiler [4] (see the snapshot presented in Figure 5), using a command line or GUI version. Although the online version is being regularly updated it has some usage limitations (e.g., input programs can only have one file) and it is mainly used to test LARA strategies applied to a program C/C++ file and in the context of demonstrations.



```
double bar() {
    return 1.0;
}

double foo() {
    double a = 0;

    for(int i=0; i<1000; i++) {
        a += bar();
    }

    return a;
}

int main() {

    foo();
```

```
aspectdef Launcher

    var flags = ["-funsafe-math-optimizations",
    "-fno-guess-branch-probability",
    "-fno-ivopts -fno-tree-loop-optimize",
    "-fno-inline-functions",
    "-funroll-all-loops",
    "-fomit-frame-pointer"];

    select function{"foo"}.call{"bar"} end
    apply
        call LibVersioningCompiler($call, "id", flags);
    end
end

aspectdef LibVersioningCompiler
```

```
#include "versioningCompiler/Utils.hpp"

extern "C" {

double bar() {

   return 1.0;
}

}


double foo() {
   std::shared_ptr<vc::Version> id = vc::createVersion("/home/specsuse
   "bar",
   {vc::Option("_", "-std=c++11")
```

```
[Warning] More than one path was found for select: function->call
    Main path: body(scope).stmt(statement).call
     2nd path: childStmt(statement).call
    Tie break: use the first path found (from a depth first search)
```

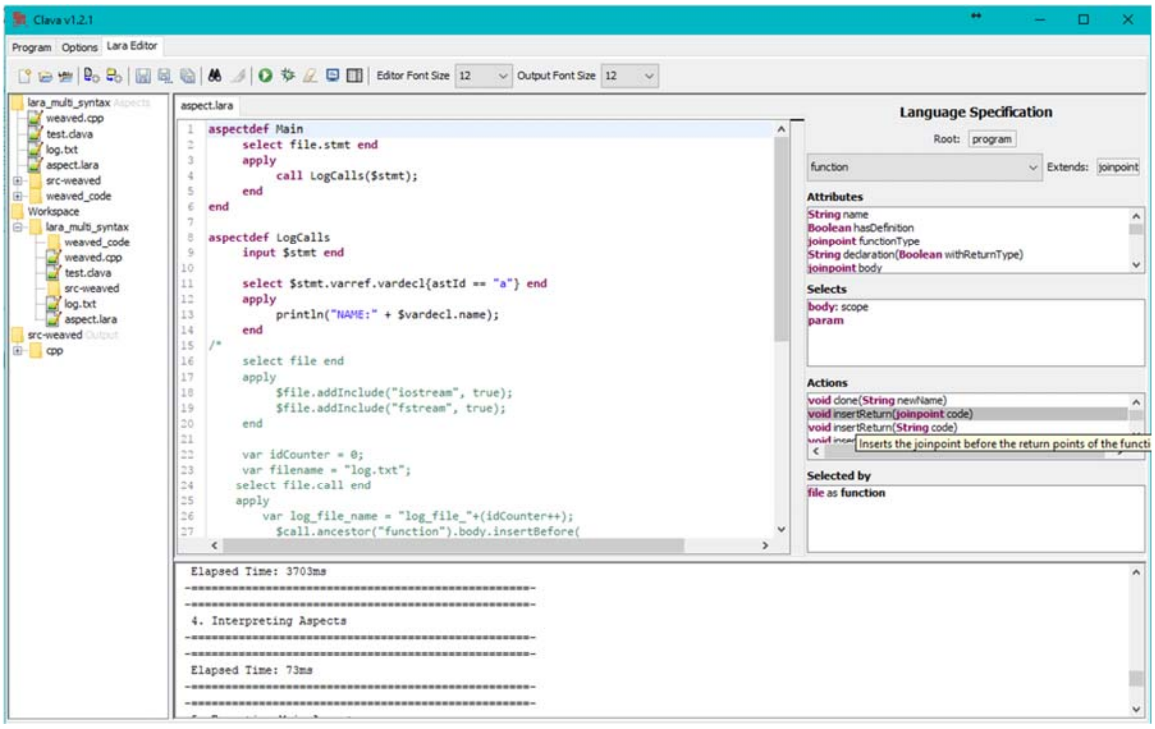
Figure 5. Snapshot of the online version of Clava.

3.4.1 GUI Version

The GUI version of `Clava` can be used by running the `CLAVA` jar file without arguments:

```
        java -jar clava.jar
```

Then the GUI presented in Figure 6 appears.



```
aspectdef Main
     select file.stmt end
     apply
          call LogCalls($stmt);
     end
end

aspectdef LogCalls
     input $stmt end

     select $stmt.varref.vardecl{astId == "a"} end
     apply
          println("NAME:" + $vardecl.name);
     end
/*
     select file end
     apply
          $file.addInclude("iostream", true);
          $file.addInclude("fstream", true);
     end

     var idCounter = 0;
     var filename = "log.txt";
    select file.call end
    apply
        var log_file_name = "log_file_"+(idCounter++);
          $call.ancestor("function").body.insertBefore(
```

```
String name
Boolean hasDefinition
joinpoint functionType
String declaration(Boolean withReturnType)
joinpoint body
```

```
body: scope
param
```

```
void clone(String newName)
void insertReturn(joinpoint code)
void insertReturn(String code)
Inserts the joinpoint before the return points of the functi
```

```
file as function
```

```
 Elapsed Time: 3703ms
-----------------------------------------------------
-----------------------------------------------------
 4. Interpreting Aspects
-----------------------------------------------------
-----------------------------------------------------
 Elapsed Time: 73ms
-----------------------------------------------------
-----------------------------------------------------
```

Figure 6. Snapshot of the GUI version of `Clava`.

Using the GUI users can create configurations for weaving projects. The GUI also provides a built-in IDE for developing LARA aspects, with syntax highlighting and checking, quick access to project files and debugging options. Configuration files can be created by using the tab `Options`, clicking on the button `Save as`… and selecting a name and location for the configuration file.

3.4.2 Command Line Version

There are two main modes in command line, either passing all arguments (LARA file, parameters, etc.), or passing a configuration file that was built with the graphical user interface, with the possibility of overriding values defined in the configuration file.

Without a configuration file:

In the case the folder where we intend to execute `Clava` includes the aspect LARA and the application source code, we can simply execute:

```
        java -jar clava.jar Aspect.lara
```

When we intend to specify a folder where all the source code is:

```
       java -jar clava.jar <aspect.lara> -p <source_folder>
```

where <aspect.lara> is the LARA aspect you want to execute, and <source_folder> is the folder where the source code is.

When we intend to specify an output folder (where all the generated code will be output) we can execute:

```
    java -jar clava.jar <aspect.lara> -p <source_folder> -o <base_folder> -of output
```

where <output_folder> is the output folder.

The generated weaved code is output to the subfolder “weaved_code” of the folder specified by -o. users can change the name of this subfolder using the flag “-of”:

```
        java  -jar  clava.jar  <aspect.lara>  -p  <source_folder>  -o  <output_folder>  -of
<weaved_code_folder>
```

where <weaved_code_folder> is the name of the subfolder.

There are more command-line options available, which can be consulted by running:

```
      java -jar clava.jar --help
```

With a configuration file:

To pass a configuration file, use the flag -c:

```
      java -jar clava.jar -c <config.clava>
```

where <config.clava> is the configuration file created with the GUI.

3.5 *Examples*

We include in:

```
     https://github.com/specs-feup/specs-lara/tree/master/ANTAREX
```

a number of representative LARA examples in the context of `Clava`. Most of them can be also tried using the online version of `Clava` [4].

The following table enumerates some of the examples available and that can be used for experimenting with LARA and Clava.

Examples:	**Link to access the code:**
Timing an application or sections of an application	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/Timer
Monitoring Energy/Power Consumptions	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/Energy
Exposing and exploring OpenMP parameters	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/OmpThreadsExplore
Applying Multiversioning	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/Multiversioning
Interfacing with mARGOt	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/MargotExamples
Interfacing with LibVersioningCompiler	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/LibVersioningCompiler

4 Conclusions

This deliverable provides the LARA front-end and `Clava`, a source-to-source C/C++ compiler responsible to output C/C++ code according to the input strategies specified in LARA. The framework proposed consists of `Clava`, a LARA frontend and an interpreter engine adapted to the semantics and constructs for runtime adaptivity strategies, and is integrated in the ANTAREX compilation flow.

5 References

[1] The LARA wiki, http:/www.fe.up.pt/~specs/projects/lara/doku.php?id=lara:downloads

[2] clang: a C language family frontend for LLVM, https://clang.llvm.org/

[3] Clava Language Specification, http://specs.fe.up.pt/tools/clava/language_specification.html

[4] CLAVA online version, http://specs.fe.up.pt/tools/clava/

[5] Java downloads, http://www.oracle.com/technetwork/java/javase/downloads/index.html

6 Acronyms and Abbreviations

Clava	-	C/C++ frontend for LLVM
AST	-	abstract syntax tree
Clang	-	source to source C/C++ compiler with LARA support for weaving
Larai	-	LARA Interpreter
Larac	-	LARA Compiler
JRE	-	Java Runtime Environment
JDK	-	Java Development Kit