Hands-on: Clava

Pedro Pinto
João M. P. Cardoso

November 3rd 2018

Abstract

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

Getting Started

Linux Preparation

1. Get the tutorial files from `specs.fe.up.pt/tutorials/PACT2018.zip`
2. Uncompress the tutorial files (the new directory will be called `<BASE>`)
3. Get the script from `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`

Windows Preparation

1. Get the tutorial files from `specs.fe.up.pt/tutorials/PACT2018.zip`
2. Uncompress the tutorial files (the new directory will be called `<BASE>`)
3. Get the jar from `specs.fe.up.pt/tools/clava.jar`
4. Get The CMake files from `https://bit.ly/2EVVnF7`
5. Uncompress the files and leave them in an accessible place

6. Open a console

7. Start Clava by running the command `java -jar clava.jar`

This should run Clava in GUI mode. In the first tab, *Program*, you will 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.

1 Call Graph

1.1 Main Idea

This first section introduces the basics of the LARA language. This example will also show 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 will select tuples of in the form $< function, call >$, and increment a counter each time a tuple is found. At the end we will print the graph in *dot* format.

1.2 Implementation

First, we will load 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 that will 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 that will be used. Right now you don't need to worry about the remaining the options.

Click the tab *LARA Editor* and you will see the LARA aspect that will be applied. The code starts with the `aspectdef` keyword, which defines a strategy. It is similar to a JavaScript function, but where you can use other LARA keywords.

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, we iterate over each of these pairs. We use normal JavaScript to keep a count of each pair we've 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!* .

2 Logging

2.1 Main Idea

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

2.2 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 should show the modified code, and the loop should now have a `printf` call right before it.

2.3 Exercise: Inserting C++ Code

Open the following configuration file:

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

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

2.4 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 creation 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 appears in the output window should have printing code before the loop that uses `printf` instead of `std::cout`.

3 Measurements

3.1 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 will 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.

3.2 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 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++.

3.3 Exercise: Measure Energy Consumption

Open the following configuration file:

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

The LARA strategy is incomplete! In this exercise you should 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/`****).*

4 AutoPar

4.1 Main Idea

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

4.2 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 will find Clava in your system and will apply the LARA strategy over the application before the compilation and linking stages. The strategy will print the modified code, please check the output and verify that the code now has OpenMP pragmas.

Finish building the application:

```
make
```

5 Exploration

5.1 Main Idea

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

5.2 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 you should 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 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 will 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 &
```