Source-to-source Compilation for Instrumentation and Code Transformations

João Bispo, Pedro Pinto

2019-05-08 – 3rd Workshop of the Green Software Lab 2019

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

INESCTEC ASSOCIATE LABORATORY PORTUGAL

SPeCS Special-Purpose Computing Systems, languages and tools

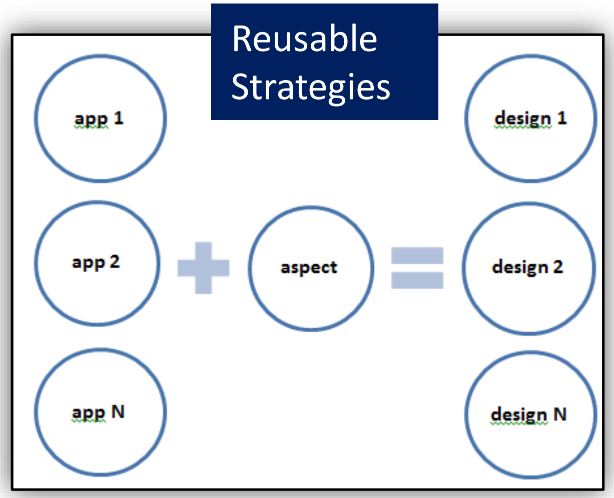
Outline

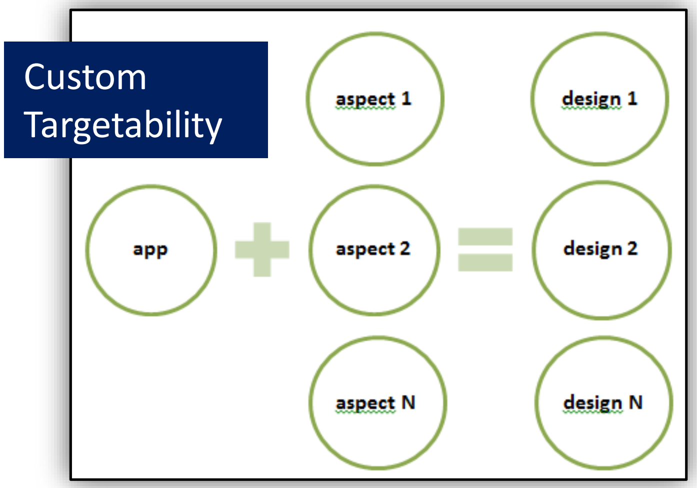
- DSL-based Source-to-Source
  - Motivation
  - Use Cases
- LARA
  - Framework
  - Language
  - Compilers
- Clava
  - Tool-flow
  - Example

Motivation

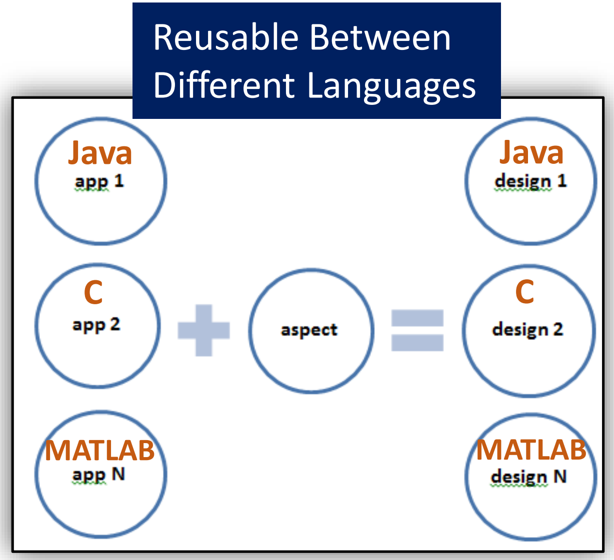
- Source-to-source compilation
  - Output code in the same language as input code (e.g., C-to-C)
  - Useful for code analysis, generation and transformation
  - Not tied to a particular compiler toolchain
- DSL-based
  - Domain-specific constructs
  - Encode strategies separately from the application
  - Enables features not present in target language, and multi-language support

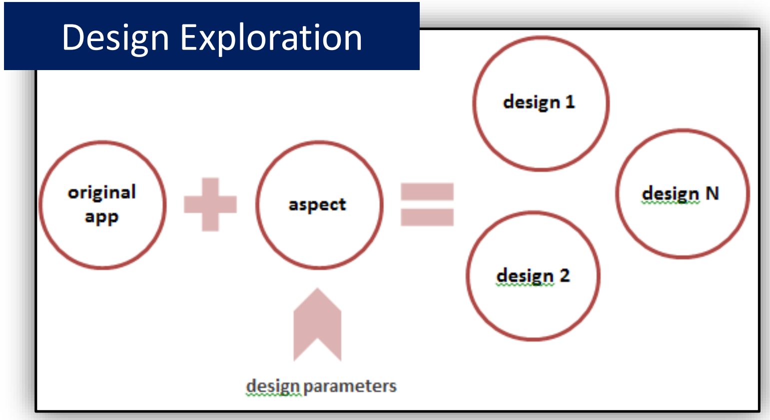
Use Cases


Reusable
Strategies
app 1
app 2
app N
aspect
design 1
design 2
design N

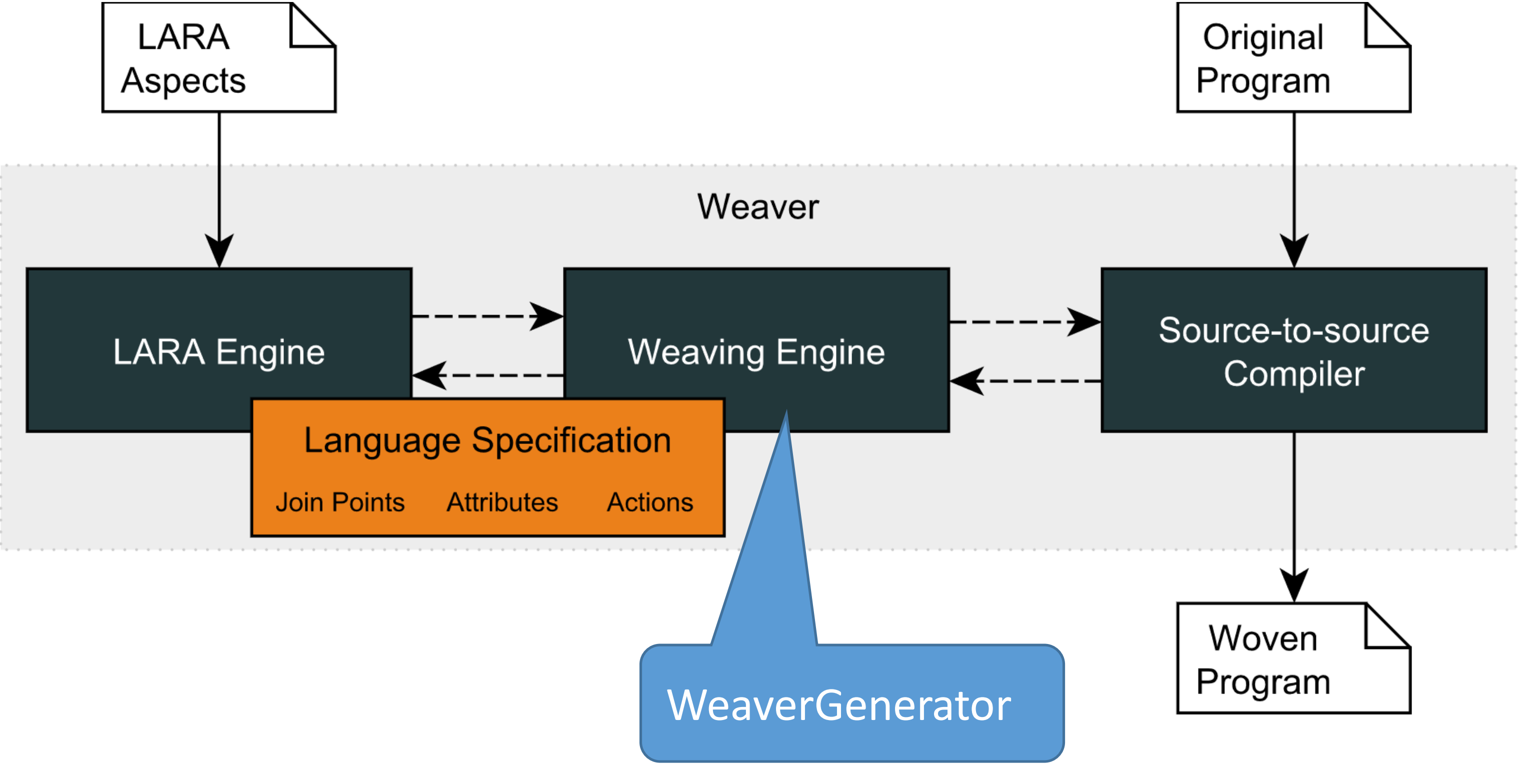

Custom
Targetability
app
aspect 1
aspect 2
aspect N
design 1
design 2
design N

Use Cases


Reusable Between
Different Languages
Java
app 1
C
app 2
MATLAB
app N
aspect
Java
design 1
C
design 2
MATLAB
design N


Design Exploration
original
app
aspect
design parameters
design 1
design 2
design N

LARA Framework



The LARA Language

- JavaScript-based language
- Strategies written separately from application logic code
- Not tied to a specific target language
  - *Weavers* binds LARA code to a target language
  - Current languages: Java, **C**, **C++** and MATLAB

```
aspectdef myAspect
  input
    in0, in1=3;
  end
  output
    out0, out1;
  end

  select … end
  apply … end
  condition … end

  function h() { }
  var z = 2;
end
```

J. M.P. Cardoso, T. Carvalho, J. G. de F. Coutinho, W. Luk, R. Nobre, P. C. Diniz, Z. Petrov, "**LARA: An Aspect-Oriented Programming Language for Embedded Systems,**" in *Int'l Conf. on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany, March 25-30, 2012.

Main LARA Features

- **Select-apply** clauses
  - **Select** points of interest in the code
  - **Apply** analysis and transformations over them

```
select method end
apply
    ...
end
```

- Modularity and reuse based on **calling aspects** and **using parameters**

```
apply
  call LoopTiling(64);
  call Timer("ns");
end
```

- **Composition of strategies** based on other strategies

LARA Select

- Access points on the source code
- Uses an hierarchical point chain
  - Defined in the language specification
- Points not present in the chain are inferred
- Filtering based on attributes

```
select file.function.body.call end
```

```
select function.call end
```

```
select function{name=="draw"}.call end
```

LARA Apply

- Iterates over the selected points (prefixed with $)
- Any point in the select statement can be accessed
- Can access point attributes
- Can change the application

insert before|after|replace

- For injecting code in input application source code

exec

- For executing a compiler action

def

- For defining the value of an attribute

```
select function{name=="draw"}.call end
apply
   $call.insert before 'code to inject';
   insert before 'more code';
end
```

```
$loop.exec interchange($innerLoop);
```

```
$var.def  type='float';
```

LARA Source-to-Source Compilers

- MATISSE: MATLAB-to-C/OpenCL compiler
  - specs.fe.up.pt/tools/matisse
- CLAVA: C/C++ source-to-source compiler
  - specs.fe.up.pt/tools/clava
- KADABRA: JAVA source-to-source compiler
  - specs.fe.up.pt/tools/kadabra
- **All tools have online demos**

MATISSE

CLAVA

KADABRA

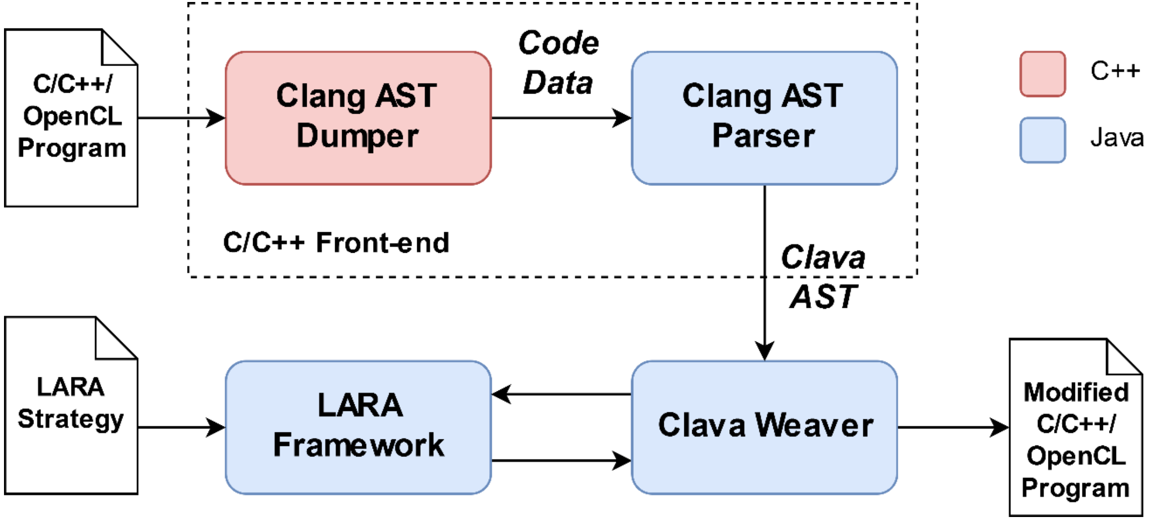
Clava

- Source-to-source C/C++/OpenCL compiler
- User-defined strategies written in LARA
- Several kinds of strategies possible
  - Analysis, Generation, Insertion, Modification
- Open-source
  - github.com/specs-feup/clava

Clava - Toolflow

- Clang-based parser

- Custom Clava AST
  - AST-based transformations



Examples

1. Static profiling
   1. Call Graph
   2. Static Report
2. Code Insertion
   1. Logging with Insertions
   2. Logging with APIs
   3. Measurements
3. Code Optimization
   1. Gprofer
   2. AutoPar
   3. Exploration
   4. Loop Tiling Exploration

Download Clava and examples:

specs.fe.up.pt/tutorials/INDIN2018.zip

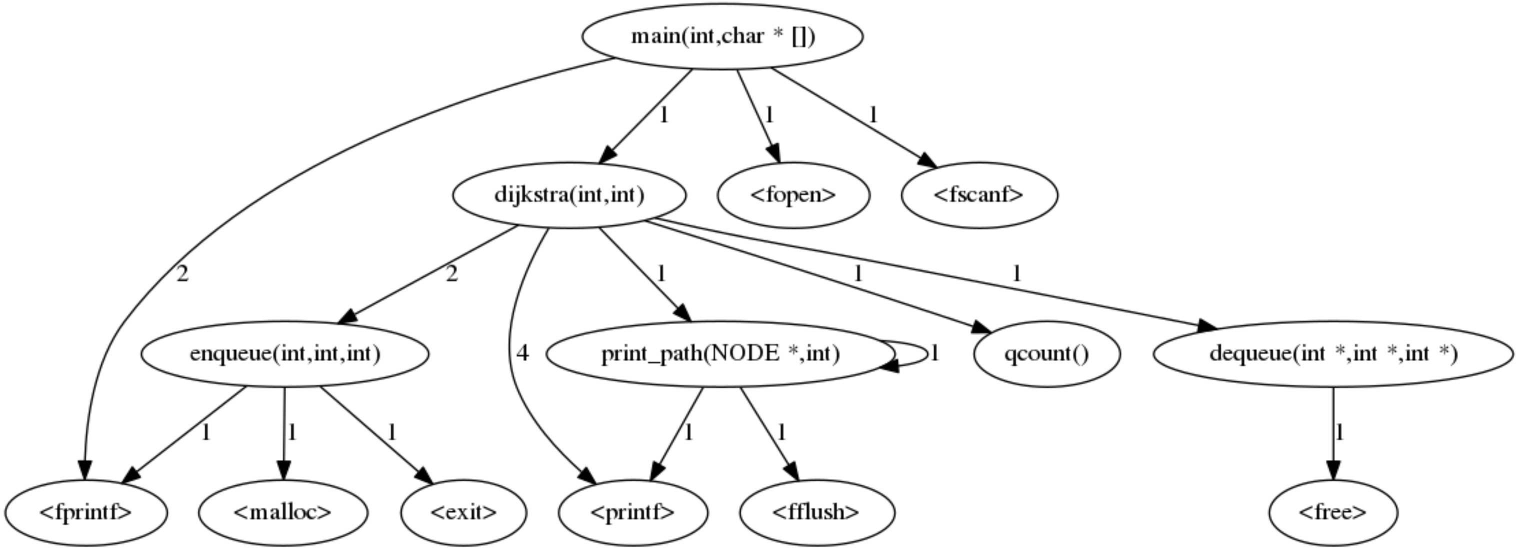
Call Graph

- Build a static call graph from the application source
- "Supergraph" of the dynamic call graph
- Edges indicate how many times a call appears in the code

- Strategy
  1. Select all methods (caller) and the calls inside (callee)
  2. Make <caller, callee> tuples
  3. Generate a graph with the tuples in dot format

Call Graph

- Test in webgraphviz.com



Static Report

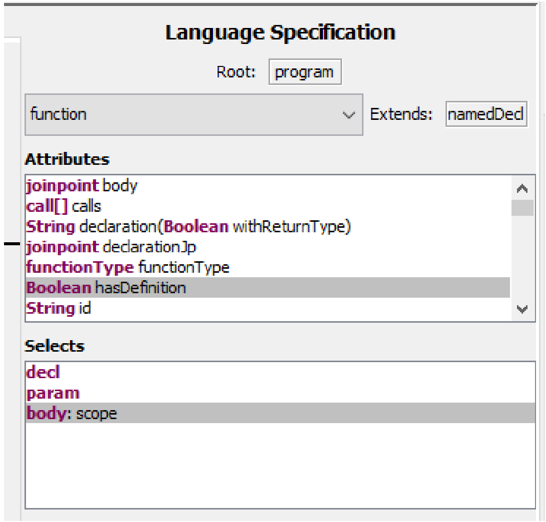
- Generate a report about the application
  - Number of files, functions and calls
  - Number and types of loops
  - Call information

- Strategy
  1. Select files, functions and calls to count
  2. Select loops and query their type
  3. Get <caller, callee> tuples
  4. Print reports

Language Specification

- Online: specs.fe.up.pt/tools/clava/language_specification.html

- IDE:



Logging with Insertions

- Log certain execution events, e.g.:
  - Start of loops
  - Entering functions

- Strategy
  1. Select loops and their parent file
  2. Insert logging code before loop
  3. Add header include at the start of the file
  4. Do the same for functions but log at the start of the body

Code Insertion with LARA

- `insert` injects literal code into the application

- Upsides:
  - extremely versatile, can insert any code you want
- Downsides:
  - cumbersome (\n), error prone, opaque

- Mitigating the downsides:
  - `codedef`
  - Clava option to verify syntax
  - `Clava.rebuild()`

Logging with APIs

- Log certain execution events (a more complex example)
- Make use of Clava APIs

- Strategy
  1. Same as before for functions
  2. Look for writes to variables inside a specific function
  3. Filter variables based on type
  4. Log when the writing happens using the Logger API

Clava Documentation

- specs.fe.up.pt/tools/clava/doc/

- clava.mpi.patterns.ScalarPattern
- clava.opencl.KernelReplacer
- clava.opencl.KernelReplacerAuto
- clava.opencl.OpenCLCall
- clava.opencl.OpenCLCallVariables
- clava.util.ClavaDataStore
- clava.util.SingleFile

LARA API

- lara.Compilation
- lara.Csv
- lara.Debug
- lara.Io

import lara.code.Logger;

Classes:

Logger

Constructor

Logger

Instance Members

Type

append()

appendChar()

appendDouble()

appendHex()

Measurements

- Collect metrics on certain events or around pieces of code
- Measure execution time and energy consumption

- Strategy
  1. Capture loops inside a specific function
  2. Call APIs to measure around the selected loops

Gprofer

- Profile an application using gprof
- This can be the start of your analysis and optimization cycle

- Strategy
    1. Import and configure Gprofer
    2. Profile the application
    3. Get hotspot and its gprof information

AutoPar

- Improve execution performance with OpenMP
- Free the user from analysis

- Strategy
  1. Select target loop based on **pragma**
  2. Call AutoPar API to parallelize the target loop

Exploration

- Perform a design space exploration on an OpenMP application
- This can be the output of AutoPar
- Automatically explore the number of threads

- Strategy
    1. Use LAT to define a variable range for the thread exploration
    2. Configure compilation options (in this case, activate OpenMP)
    3. Define the scope where LAT will perform changes
    4. Define the scope where LAT will collect metrics
    5. Start the exploration

Loop Interchange Exploration

- Apply loop interchange to a matrix multiplication kernel
- Automatically explore what the best permutation is
- Use some more Clava APIs

- Strategy
  1. Generate all possible interchange permutations and for each:
     1. Apply interchange
     2. Add code to measure execution time
     3. Compile and execute the application
     4. Save the results
  2. Print the results

Conclusions

- Clava is a **source-to-source** C/C++ compiler
- **Strategy reusability** between programs and languages
- **Fine-grained**, structural/syntactic points with **semantic information**
- Code analysis, generation, insertion, and modification

Backup Slides

The LARA Language

- Join Point Model
  - Allows the front-end to adapt to other target programming languages
- Attribute Model
  - Allows LARA to access join point values and to associate values to join points
- Action Model
  - Allows LARA to express actions

Join Point Model

```
|\_var
|\_declaration
 \_function
      |\_prototype
       \_body
           |\_first
           |\_last
           |\_var
           |\_call
           |\_if
           |   |\_condition
           |   |\_then
           |    \_else
                \_loop
             |\_init
                 |\_condition
                 |\_counter
                 |\_body
                  \_control
```

Attribute Model

```
|\_var
|    |\_name
|    |\_type
|    |\_is_array
|    |\_is_pointer
|    |\_is_write
|    |\_is_read
|    |\_is_in
|     \_is_out
|\_function
|    |\_name
|    |\_num_lines
|     \_return_type
|\_call
|    |\_name
|    |\_return_type
|    |\_num_argin
|     \_num_argout
 \_loop
     |\_type
     |\_is_innermost
     |\_num_iterations
     |\_increment_value
     |\_rank
      \_nested_level
```

Instrumentation Example: Static Call Graph

- Select all pairs of <caller, callee> function tuples
- Increments a counter every time the same tuple is observed
- Uses this counter to print the static call graph in *dot* format
- Useful to check the structure of the code
  - Takes into account all possible function calls

```
aspectdef StaticCallGraph
    var cg = new LaraObject();
    select function.call end
    apply
        cg.increment($function.name, $call.name);
    end
    println('digraph static_cg {\n');
    for (f in cg) {
        for (c in cg[f]) {
            print(f + '->' + c);
            println(' [label="' + cg[f][c] + '"];');
        }
    }
    println('}');
end
```

AOP Approach

- Several AOP languages
- No reusability between AOP languages
- Flexibility on the join point capture
- Include the support of code transformations

Concerns related to code transformations and compiler optimizations:

- Performance, Power, Energy
- Parallelism, Concurrency
- Monitoring, Test, Debug
- Safety, Security
- Targeting hardware accelerators, multicore and manycore architectures
- Different tool flows
- Fully explore compiler optimizations

Existence of many crosscutting concerns!