H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

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

Deliverable D2.3: Compiler and Code Generation Support

Horizon 2020
European Union funding
for Research & Innovation

Deliverable Title:	Code Refactoring Tool Guided by DSL		
Lead beneficiary:	INRIA (France)		
Keywords:	DSL, library, accompany report		
Author(s):	Erven Rohou (INRIA); Imane Lasri (INRIA); Giovanni Agosta (POLIMI); Gianluca Palermo (POLIMI), Stefano Cherubin (POLIMI), João Bispo (UPORTO); Pedro Pinto (UPORTO); João M.P. Cardoso (UPORTO); Ricardo Nobre (UPORTO)		
Reviewer(s):	Cristina Silvano (POLIMI)		
WP:	WP2	**Task:**	T2.3
Nature:	Other	**Dissemination level:**	Public
Identifier:	D2.3	**Version:**	V1.0
Delivery due date:	August 30th, 2017	**Actual submission date:**	Sept. 27th, 2017

Executive Summary: This document is the accompanying report associated with the **Deliverable D2.3**. It represents the results of the activities carried out by the project partners from M09 to M24 in **Task 2.3 "Compiler Technology and Code Generation"** under the leadership of INRIA. **D2.3** delivers compiler technology able to generate multiple versions of the same function and to provide multiple operating points. These multiple operating points can then be selected by users or by an autotuner. The software delivered as **D2.3** combines source-to-source analyses and transformations driven by the LARA DSL, compiler-level optimizations and controlling agents, as well as techniques dedicated to custom/adaptive precision.

Approved and issued by the Project Coordinator: [illegible]	**Date:** **Sept. 27th, 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

This document is the accompanying report, describing the software delivered as part of **D2.3**. The document has intentionally kept short, the documentation, installation and user guides are hosted with the code they refer to.

This work is delivered under the leadership of INRIA, with contributions from POLIMI and UPORTO related to Task **T2.3**. The purpose of this task is to deliver the compiler technology upon which autotuning will be built. The key idea is to be able to produce multiple instances of the same software, i.e. versions of the program that are functional identical, but differ in non-functional aspects. This is done at several levels. At high-level, a source-to-source approach, lets programmers guide the autotuner through the use of the LARA DSL. At lower-level a compiler engine is more appropriate to generate multiple versions, by enabling/disabling its optimizations, or changing their order. We also proposed to rely on custom precision to propose alternative operation points.

Deliverable **D2.3** combines a set of compiler-related technologies that, together, achieve the goal of Task **T2.3**. The contributions are all part of an advanced compilation approach that target multiple aspects and compilation layers:

- Code specialization via multiversioning and custom fixed-point data types;
- Function specific compiler optimizations via flag selection and phase selection and ordering;
- Split compilation by considering the recompilation of certain functions at runtime;
- Runtime autotuning and the integration of the autotuning in the input application code.

All the support presented in this deliverable strongly contribute to the ANTAREX tool flow and provide a unique and advanced compilation approach.

The contributions are reported as following.

Section 2 introduces the code specialization strategies. This includes first the general source-to-source support provided by LARA and Clava (our C/C++ source to source compiler); we then illustrate specialization with the case of computation with fixed-point representation; and finally consider a more general case of approximate computing based on the if-memo tool and on the use of memoization. The latter tool is extended in two directions to fit the purpose of ANTAREX: integration with LARA, as well as standalone version that supports very fast approximation of numbers in floating point representation.

Section 3 describes the ANTAREX support for selection and phase ordering of compiler optimizations in the context of the GCC and LLVM compilers. The support is provided by LARA aspects and we include a Simulated Annealing based scheme for design space exploration (DSE). The LARA aspects provided can be easily modified to support other compilers and/or to implement other DSE schemes.

Section 4 describes the integration of the mARGOt autotuner in our target applications. It covers code generation and configuration generation and describes the related LARA aspects.

Section 5 presents the implementation of the split-compilation approach, describing the approach and the LARA aspects designed for the split-compiler.

Section 6 concludes this report.

Even though Sections 2.1 and 5 describe work related to multiversioning, we chose to present them in different sections because they fulfil different purposes.

2 Code Specialization

2.1 LARA and CLAVA support for multiversioning [UPORTO]

We have developed a set of actions in the Clava weaver that enable easier multiversioning. Since all the required code is automatically generated by Clava using LARA and it is heavily application-specific, the methodology is implemented using LARA aspects that express strategies for applying code multiversioning.

As part of research during exploration of the Polybench benchmark suite, we developed a set of aspects in a strategy that generated multiple versions of the kernel of each benchmark, each with a different name. These versions were then changed to include different OpenMP parameters in their loops, controlling the binding policy and the number of threads used. Furthermore, each version was compiled with a different set of optimizations as they were surrounded with compiler pragmas controlling the optimization level. The cloned versions can be further altered in several ways, including: different framework parameters, different algorithms or parameterizations of those algorithms and even different code transformations.

The developed aspects also generated a wrapper and changed all calls to the original target to call the wrapper instead. The wrapper, according to the state defined by the global control variables, decides which version to run. The initial control values are defined through a command line interface and then they can be changed during runtime as the developer sees fit.

The developed actions include the declaration and definition of global variables (useful for controlling which version is called) and the cloning of functions of interest which can then be changed to suit the needs of the developer. We are considering extending Clava with actions to facilitate code generation and insertion of function wrappers that can be used to switch versions at runtime, according to control variables.

As part of the described work, we also had a LARA aspect to generate and insert code for the C/C++ interface of mARGOt, which was then used to tune the control variables and, in turn, choose which version to run according to the desired goal.

The LARA aspects are publicly available in the repository: https://github.com/specs-feup/specs-lara, located inside the *ANTAREX/Multiversioning/lara* directory.

The multiversioning strategies are not integrated with Clava as libraries that are distributed with the compiler because they are focued on a very specific concern and can hardly be generalized. Nevertheless, the generic parts have been made available through Clava actions which can be used in other LARA aspects.

To test the multiversioning strategy, please follow the instructions provided in the file *multiversion-script.pdf*, which can be found at: https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P. This script will guide you through the download of the weaver and all needed code, the execution of the weaver and the generation of final application with several versions that can be called at runtime.

2.2 Custom precision with fixed-point representation [INRIA+POLIMI]

We explored the benefits of computations in fixed-point representation, with a focus on not losing vectorization capability of the compiler due to the presence of the fixed-point type. Since the fixed-point is not a standard C-type, it has to be represented somehow, but the compiler loses the semantics associated to the type. This work is achieved thanks to the integration of several techniques.

Fixed-point representations are typically used in hardware design, where the width can be arbitrarily chosen for each value, on a per-bit basis. Converting an application from floating-point to fixed-point representations is a sophisticated process. Since the widths of the integer and fractional parts are fixed and pre-computed, they must be carefully chosen to limit loss of precision. This is accomplished for a given computation by assessing the dynamic range (minimum and maximum) of its input values, and propagating these ranges through all intermediate values – in a data-flow manner – to the results. Based on all ranges, an appropriate fixed-point representation is selected that minimizes added noise.

When using general-purpose processors, on the contrary, the actual bit-widths are constrained by the underlying hardware, typically the width of registers. In practice, such containers are 16-bits, 32-bit or 64-bit wide. Still, the cost of floating point arithmetic, even in optimized hardware implementations, is high enough that it is worth investigating the benefits of fixed-point operation even in the context of high performance computing. In our work, we consider the x86-64 architecture with Intel MMX, Intel SSE4.2 and AVX2 vector extensions enabled. We exploit vectorized fixed point operations to speed up the execution, at the expense of some loss in precision. This loss in precision can actually be reduced if the range of values processed is small enough.

2.2.1 GeCoS

The GeCoS (Generic Compiler Suite) project is an open source compiler infrastructure developed at Inria in the CAIRN research group. GeCoS is a C compiler infrastructure entirely written in Java following Model Driven Engineering design principles. In particular, GeCoS leverage on the Eclipse Modeling Framework (EMF). GeCoS is mainly a Source to Source compiler targeting High Level Synthesis tools, but also automatic parallelization for heterogeneous embedded multi-cores.

In the context of custom fixed-point data types, we use the GeCoS source-to-source compiler to first perform a value range propagation analysis to propagate the value range information from annotated variables along data-dependence chains, thus inferring the value range for each variable involved in the computation. The output of this analysis is a fully annotated C/C++ source code having the range of values each variable can assume annotated on its declaration. To perform the value range propagation analysis, we repurposed GeCoS and developed our own plugin.

From the value ranges, it is possible to compute the number of bits needed for the representation of the integer part of the fixed-point representation. The width of the fractional part is then obtained as the difference between the architectural constraint on the total bit size and the size of the integer part.

In a second step, GeCoS takes then care of replacing the annotated floating point variables with their fixed-point equivalent. It also adds to the original source code the utility functions to perform data type conversions from floating-point to fixed-point and vice versa. The output of this stage is a new version of the kernel source code exploiting fixed-point computation instead of floating point computation. The fixed point code can then be compiled as a standard C/C++ source file using GCC or Clang/LLVM.

GeCoS is available at the following URL: http://gecos.gforge.inria.fr.

To ease the source-to-source compilation we developed a C++ library which implements fixed-point logic. By using our library, the compiler (GCC or Clang) is able to handle fixed point numbers as a user-defined data type. We defined and implemented data type conversion routines among arithmetic operators. We make sure in our implementation that the internal representation will be interpreted by the compiler as the shortest integer data type available which is able to contain the required number of bit for the integer and fractional. The number of integer and fractional bits are specified as C++ template parameters in the variable declaration.

The library is available on github: https://github.com/skeru/fixedpoint.

The plugin is hosted on the GeCoS repository: https://gforge.inria.fr/projects/gecos, in the branch `/trunk/gecos/core-`

`emf/fr.irisa.cairn.gecos.model.analysis/src/fr/irisa/cairn/gecos/model/analysis/profiling`.

2.2.2 Example

In a first step, the programmer shall describe which variables are candidates for conversion to fixed-point representation. This is achieved is a very user-friendly way by annotation the code with a C/C++ pragma, as shown below. The code is then automatically processed by the source-to-source compiler to keep track and the dynamic range of the selected variables and to emit the minimum and maximum values encountered at the end of execution.

```
#pragma VARIABLE_TRACKING variable
for (int i=0, i<10, i++) {
  variable = i;
}
output:
variable_min = 0
variable_max = 9
```

```
Program output:
variable_min = 0
variable_max = 9
```

Based on the dynamic range, it is also possible to select the best fixed-point representation, i.e. choosing the size of integer and fractional parts. The integer part must be large enough to avoid overflows, but this is at the cost of the fractional part, and thus precision. The code excerpts below illustrate original and final transformed code.

```
#define SIZE1 10
#define SIZE2 10
#pragma VARIABLE_TRACKING m tmp foo
double m[SIZE1][SIZE2];
double tmp;
double foo;
foo = 0;
for (size_t i = 0; i < SIZE1; ++i) {
 for (size_t j = 0; j < SIZE2; ++j) {
  if (m[i][j] > m[j][i]) {
   foo = foo + m[i][j] * m[j][i];
   tmp = m[i][j];
   m[i][j] = m[j][i];
   m[j][i] = tmp;
  }
 }
}
```

```
#define SIZE1 10
#define SIZE2 10
double m[SIZE1][SIZE2];
FixedPoint<3,29> m_fixp[SIZE1][SIZE2];
double tmp;
FixedPoint<3,29> tmp_fixp;
double foo;
FixedPoint<8,24> foo_fixp;
convert2DtoFixP<double, SIZE1, SIZE2>(m,m_fixp);
foo_fixp = 0;
for (size_t i = 0; i < SIZE1; ++i) {
 for (size_t j = 0; j < SIZE2; ++j) {
  if (m_fixp[i][j] > m_fixp[j][i]) {
   FixedPoint<8,24> _s2s_tmp_foo_0;
   _s2s_tmp_foo_0 = m_fixp[i][j] * m_fixp[i][j];
   foo_fixp = foo_fixp + _s2s_tmp_foo_0;
   tmp_fixp = m_fixp[i][j];
   m_fixp[i][j] = m_fixp[j][i];
   m_fixp[j][i] = tmp_fixp;
  }
 }
}
convertScalarToFloat<double>(foo_fixp,foo);
convert2DToFloat<double,SIZE1,SIZE2>(m_fixp,m);
```

2.3 Memoization using LARA and if-memo [UPORTO + INRIA]

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. The transformation is implemented as a particular specialization, where candidate functions are redirected to a wrapper that takes care of the lookup.

In the ANTAREX project, we have two different approaches for memoization, one that uses LARA to instrument the code and build a memoization table based on profiling data, and another that intercepts calls to libraries and generates the table online, with the results of the called functions.

2.3.1 LARA-based Memoization

The current LARA memoization approach for ANTAREX addresses instrumentation and code adaptation by using LARA aspects. There are three main phases in this strategy. First, an instrumentation step injects code to collect profiling information. Then, a second step compiles and executes the instrumented application to generate profiling data. Finally, a table generation step uses the collected profile, builds a table and logic code and inserts it in the application code. The overall tool flow is presented in Figure 1.

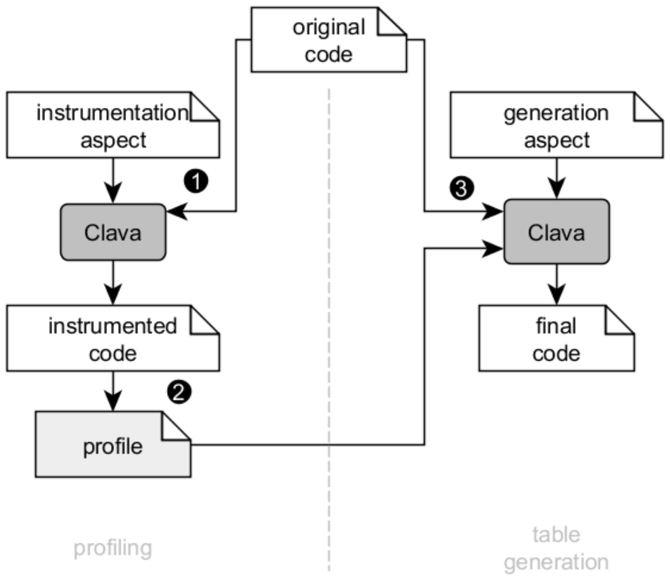

Figure 1 - Overall tool flow of the LARA-based memoization approach.

The instrumentation part of this technique is rather simple. We rely on a memoization profiling library that has been developed for C. Guided by LARA, Clava is able to insert calls to this library at the call sites where one of our target functions is called. The library collects the values of inputs and their frequency, as well as the corresponding outputs. At the end of the execution, another call to the library generates a report in a JSON file. An example of such a report is presented in Figure 2. In order to generate cleaner code, a LARA action, called *wrap*, generates a wrapper function for each of our target functions and replaces original calls to calls to the wrapper. Then, the instrumentation is performed at a single location for each target, inside the corresponding wrapper.

```
{
    "name" : "log",
    "elements" : "610",
    "calls" : "6538750",
    "hits" : "6538140",
    "misses" : "610",
    "counts" : [
        {
        "key" : "405c606020000000",
        "output" : "4136505a209b8279",
        "counter" : "61018"
        },
        {
        "key" : "405c80f6c0000000",
        "output" : "41369d9340dfee35",
        "counter" : "90"
        },
        ...
    ]
}
```

Figure 2 - Example of a JSON profile for memoization.

The table-building strategy loads the generated report and tries to fit as many entries as possible inside a table. Each entry is one of the elements from the *counts* array presented in Figure 2. This table, for now, has a limit of 65,536 elements and is indexed by 16 bit numbers, computed from the bits used for the internal representation of the inputs. Whenever two entries can collide using our hash function, we ensure that we keep the one that was used more frequently on the profiling run, which usually results in better performance at the end. After this table is built, Clava generates the code and injects it in the application. We use wrappers once again, so Clava only injects the table and its logic once per target, at the corresponding wrapper. An example of the generated code is shown in Figure 3.

```c
// ...
double log_wrapper( double __x )
{
    double result;
    static const uint64_t table[256][2] = {
        {0x4057c35480000000, 0x412a35411a0dfa46},
        {0x40688a03a0000000, 0x415cdc56e73c6370},
        {0x40588d77e0000000, 0x412ce8888b74fa1f},
// ...
```

Figure 3 - Example of generated wrapper and table for a log function wrapper.

The LARA aspects and supporting C code for memoization are publicly available. The LARA aspects can be found at: https://github.com/specs-feup/specs-lara, located inside the directory *ANTAREX/Memoization/lara*. The C profiling library that is used in the profiling step can be found at: https://github.com/specs-feup/specs-c-libs, located inside the *MemoiInstrumenter* directory. The memoization strategies are distributed as independent aspects, separated from Clava.

To test these memoization strategies, please follow the instructions provided in the *memoi-script.pdf*, which can be found at: https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P. This script will guide you through the download of the weaver and all needed code, the execution of the weaver, the profiling of the original application, and, finally, the generation of the memoization table.

2.3.2 Custom Precision using Memoization

We also explored another variation of if-memo that results in custom precision. When checking whether a value is already present in the lookup tables, we mask out the least significant bits of the operand. The benefit comes from two factors: repetition happens more often; and fewer values are stores which improves the behaviour on capacity and conflict.

This is implemented as a straightforward modification of the original code base: before lookup, the argument is truncated by masking off a predefined number of bits. The code is hosted in a dedicated branch of the if-memo repository:

```
svn checkout
   https://scm.gforge.inria.fr/anonscm/svn/if-memo/if-memo/branches/approx
```

3 Compiler sequences [UPORTO + POLIMI]

Compiler users tend to rely on standard optimization levels, typically represented by flags such as GCC's or Clang's `-O2` or `-O3`, when using optimizing compilers. These flags represent fixed sequences of analysis and transformation compiler passes, also referred to as compiler phase orders. An application compiled with these flags tends to outperform the unoptimized equivalent. However, there are often other assembly/binary representations of the source code in the solution space which further improve the application (e.g., higher performance, smaller code size, energy savings) when compared with using standard optimization levels [1,2,3,4,5].

There are a number of scenarios that benefit from specialized compiler sequences, as there is potential to achieve considerable performance, energy or power improvements in comparison with what is achieved with the standard optimization levels. Domains such as embedded systems or HPC tend to prioritize metrics such as energy efficiency that typically receive less attention from the compiler developers, so these domains benefit further from these specialized sequences [6].

In our Design-Space Exploration (DSE) framework, exploration schemes, objective functions, and target-specific exploration and configuration parameters are programmed with the LARA aspect-oriented programming language. Modularity is assured by the fact that a new DSE infrastructure component (e.g. new objective function) can be paired with other components from other types (e.g. available targets and available exploration schemes). Instead of relying on code annotations, code transformations and compiler mapping strategies are described in a separated file; allowing the reuse of theses transformation/mapping specifications across different sources/applications and/or targeting multiple platforms/requirements using a single annotation-free source code.

We provide two examples, one for phase selection and another for phase selection and ordering, which can be found at:

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

3.1 *Phase selection and ordering (in the context of LLVM)*

We present a Simulated Annealing (SA)-based approach targeting a system representative of an ANTAREX HPC node, using the LLVM Optimizer tool opt, which optimizes LLVM assembly IR generated by the Clang frontend, and the x86-64 backend (with `-march=native` flag) of the LLVM Static Compiler (llc). The selection of the compiler passes and their execution order within the LLVM framework is accomplished by passing the flags representing the passes in a specific order to the LLVM Optimizer. After the LLVM IR is transformed by the execution of the compiler passes in the requested order, the DSE system calls the LLVM static compiler, to generate assembly code, which is linked with the Clang linker targeting the same architecture. Figure 4 presents the compilation flow used in our approach.

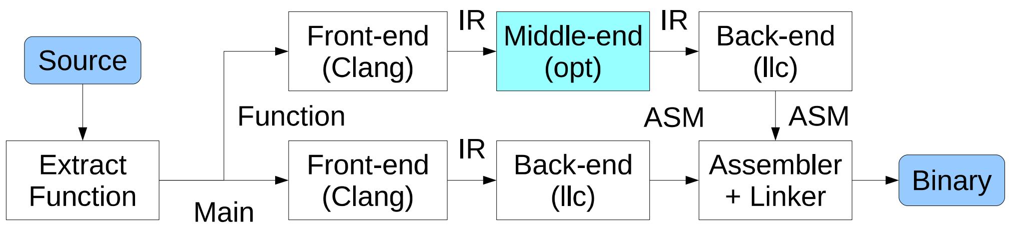

Figure 4 – Compilation flow for phase selection and ordering.

Our work in [7] proposes MiCOMP approach for Mitigating the Compiler Phase-ordering problem of the LLVM's highest optimization level using optimization sub-sequences and machine learning: an autotuning framework to eectively mitigate the compiler phase-ordering problem based on machine-learning techniques. The idea is to cluster the optimization passes of the LLVM's O3 setting into different clusters to predict the speedup of the complete-sequence of all the optimization clusters instead of facing with more than 60 different individual optimization passes. The predictive model uses (1) platform-independent dynamic features, (2) an encoded version of the compiler sequence and (3) an exploration heuristic to tackle the problem. Experimental results using the LLVM compiler framework and the Cbench suite show the effectiveness of the clustering and encoding techniques to application-based reordering of passes, while using a number of predictive models. We perform statistical analysis on the prediction space and compare against (1) standard optimization levels O2 and O3, (2) random iterative compilation, and (3) two recent non-iterative approaches. We demonstrate that our proposed methodology outperforms the performance of -O1, -O2, and -O3 optimization levels in just a few iterations, reaching an average performance speedup of 1.31 (up to 1.51) on the Cbench benchmark suite.

3.2 *Phase selection (in the context of GCC)*

We present a Simulated Annealing (SA)-based approach targeting a system representative of an ANTAREX HPC node, using the GCC compiler. In this work, the sequences of compiler passes found by the SA algorithm are normalized, so that two found sequences with the same set of passes have the same order. Then, flags corresponding to these passes are used when calling GCC.

Our work in [8] proposes COBAYN a compiler autotuning framework using Bayesian networks, an approach for a compiler autotuning methodology using machine learning to speed up application performance and to reduce the cost of the compiler optimization phases. The proposed framework is based on the application characterization done dynamically by using independent microarchitecture features and Bayesian networks. The work also presents an evaluation based on using static analysis and hybrid feature collection approaches. In addition, the work compares Bayesian networks with respect to several state-of-the-art machine-learning models. Experiments were carried out on an ARM embedded platform and GCC compiler by considering two benchmark suites with 39 applications. The set of compiler configurations, selected by the model (less than 7% of the search space), demonstrated an application performance speedup of up to 4.6× on Polybench (1.85×on average) and 3.1× on cBench (1.54× on average) with respect to standard optimization levels. Moreover, the comparison of the proposed technique with (i) random iterative compilation, (ii) machine learning–based iterative compilation, and (iii) non-iterative predictive modelling techniques shows, on average, 1.2×, 1.37×, and 1.48×speedup, respectively. Finally, the proposed method demonstrates 4×and 3×speedup, respectively, on cBench and Polybench in terms of exploration efficiency given the same quality of the solutions generated by the random iterative compilation model.

4 Integration of Autotuning using LARA [POLIMI + UPORTO]

We have developed two LARA libraries to support the integration of the mARGOt autotuner in our target applications. Currently, we support via LARA the high-level interface exposed by mARGOt. Note, however, that the decision on which feature of the autotuner internals will be supported is ongoing in Task **T3.2**. The first library deals with code generation and insertion for the interface that needs to be in the target application. The second allows for a quicker and more fluent definition of the configuration file needed by mARGOt.

4.1 mARGOt Code Generation

LARA allows an easy selection of points of interest in the code (e.g., loops and function calls) and the interaction with those points, namely through transformation actions and insertion of additional code. In order to extend an application and provide it with autotuner capabilities, we can select target points and insert code that uses mARGOt's interface. However, instead of relying on code insertion using directly LARA insert actions, we implemented a library that can be used to automatically insert the needed code with minimal configuration.

As shown in **Error! Reference source not found.**, we can just import the library and instantiate a new *Margot* object. Since the code is almost the same, except for the autotuning *block* and its target variables, we can configure these options at the start. These are the arguments to the *Margot* object constructor.

```
import antarex.margot.MargotCodeGen;
// ...
var m = new Margot('block1', 'alpha', 'beta');
```

Figure 5 - Importing and using the mARGOt code generation LARA library.

After in the aspect we can use the provided interface to generate and insert the code. For instance, **Error! Reference source not found.** shows the difference between inserting a call to the *log* function of mARGOt using this provided interface or doing it with direct calls to LARA insert actions. Besides typing less LARA code, this has the additional advantages of being less prone to human error and being more resilient to future changes in the mARGOt API. For instance, if the API is changed and *log* is renamed to *logging*, the LARA aspect is still remain valid and Clava would generate the correct code after an update to the internal library.

```
select call{'kernel_2mm'} end
    apply
        // ...
        // we can use the interface of this LARA library
        m.log($call);

        // or we can perform the insertion manually
        $call. insert after 'margot::block1::log();'
        // ...
end
```

Figure 6 - Comparison between using the library or using LARA insert actions directly.

The interface we provide with this LARA library usually expects a join point and inserts the correct code around it, as needed. Furthermore, we provide both high-level and low-level functions. The former perform more actions at once (e.g. update and log), but give less control over where the calls are inserted, since they take a single join point. The latter deal with individual actions, but provide greater control over the location of each inserted call.

The LARA aspects are publicly available in our repositories. Since they are fully integrated with Clava, the LARA library aspects can be found at: https://github.com/specs-feup/specs-lara, located inside the directory *ANTAREX/AntarexClavaApi*. You can also find some usage examples of this library in the same repository, under the directory *ANTAREX/MargotExamples/CodeGen*.

To test this library, please follow the instructions provided in the *codegen-script.pdf*, which can be found at: https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P. This script will guide you through the download of the weaver and all needed code, the execution of the weaver and the weaving of the application to include calls to mARGOt's API.

4.2 mARGOt Configuration Generation

To further help the development of applications extended with mARGOt, we have also developed a LARA library to help users to define the configurations needed by mARGOt and generate the corresponding XML code, which can be printed to the console or to a file.

Figure 7 shows how to import and instantiate the library. This figure also shows how to define a *block*. Every other actions are performed on a block. These include defining goals, knobs and monitors (Figure 7 shows the definition of a timer monitor). Although not every elements of the mARGOt configuration are currently handled, one can already generate useful configuration files.

```
import antarex.margot.MargotConfig;
// ...
var mc = new MargotConfig();
var mb1 = mc.newBlock('foo');
// ...
var tm = mb1.newTimeMonitor('my_elapsed_time_monitor');
tm.expose('avg_computation_time',
MargotExpose.AVERAGE);
tm.useMicro();
```

Figure 7 - Example of importing and using the configuration library

The LARA aspects are publicly available in our repositories. Since they are fully integrated with Clava, the LARA library aspects can be found at: https://github.com/specs-feup/specs-lara, located inside the directory *ANTAREX/AntarexClavaApi*. You can also find some usage examples of this library in the same repository, under the directory *ANTAREX/MargotExamples/Config*.

To test this library, please follow the instructions provided in the *config-script.pdf*, which can be found here: https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P. This script will guide you through the download of the weaver and all needed code and the execution of the weaver, which will generate the configuration file. This aspect will not weave any source code.

5 Split-Compilation with the LibVersioningCompiler [POLIMI+UPORTO]

We have developed a strategy and supporting code to enable the integration of libVersioningCompiler in LARA aspects. This strategy shows how we can capture calls to target functions and change the surrounding code to add support for runtime compilation. In the developed strategy we replace calls to the original function with calls to a new symbol, which is compiled and loaded on demand, as soon as we enter the function that is calling our target. This compilation uses compiler-specific flags, which are provided by the user in the LARA aspect, allowing for great control of possibly multiple versions.

The LARA aspect for libVersioningCompiler support in our repository. The LARA aspect can be found at: https://github.com/specs-feup/specs-lara, located inside the directory *ANTAREX/ LibVersioningCompiler*.

In order to use the strategy that enhances an application with support for runtime compilation of target functions, please follow the instructions provided in the *lib-script.pdf*, which can be found here: https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P. This script will guide you through the download of the weaver and all needed code, and the weaving of the aspect that inserts calls to libVersioningCompiler's API.

6 Conclusions

This document describe the software produced by Task **T2.3 "Compiler Technology and Code Generation"** of the ANTAREX project, and delivered as **D2.3**. This software combines high-level source-to-source tools, as well as lower level components dedicated to compiler technology. They are summarized below, as well as background technology brought to the consortium by some partners. Ongoing work consist in further integrating all these components to validate the overall approach of the ANTAREX approach. This will be carried on by workpackage **WP3**, as well as in the context of the two use-cases in **WP4** and **WP5**.

6.1 *Summary of background technology*

Name	Source	Description
GeCoS	INRIA	GeCoS (Generic Compiler Suite) project is an open source compiler infrastructure, written in Java following Model Driven Engineering design principles. In particular GeCoS leverage on the Eclipse Modeling Framework (EMF).
If-memo	INRIA	If-memo is a tool that intercepts calls to pure functions implemented in shared library. It caches the results of previous invocations and returns the cached results in future calls, thus saving the cost of the computation.
LARA	UPORTO	LARA is an aspect-oriented programming (AOP) language, specifically designed for allowing developers to program code instrumentation strategies, to control the application of code transformations and compiler optimizations, and to effectively control different tools in a toolchain. LARA provides a separation of concerns, including non-functional requirements and strategies, while allowing to exploit the benefits of an automatic approach for various domain-specific and target component-specific compilation/synthesis tools.

6.2 *Summary of foreground technology, delivered as D2.3*

Name	Description	Location
LARA and Clava multiversioning	Sec. 2.1	https://github.com/specs-feup/specs-lara https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P
Conversion to fixed-point	Sec. 2.2	https://github.com/skeru/fixedpoint (svn) https://gforge.inria.fr/projects/gecos/trunk/gecos/core-emf/fr.irisa.cairn.gecos.model.analysis/src/fr/irisa/cairn/gecos/model/analysis/profiling
Memoization with if-memo and LARA	Sec. 2.3.1	https://github.com/specs-feup/specs-lara https://github.com/specs-feup/specs-c-libs https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P

Modified if-memo for approximate memoization	Sec. 2.3.2	(svn) https://scm.gforge.inria.fr/anonscm/svn/if-memo/if-memo/branches/approx
Phase selection and ordering (LLVM) Flag selection (GCC)	Sec. 3.1 Sec. 3.2	https://github.com/specs-feup/specs-lara, located inside the directory ANTAREX/ PhaseSelectionAndOrdering.
Autotuning with LARA	Sec. 4	4.1 mARGOt code generation https://github.com/specs-feup/specs-lara https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P 4.2 mARGOt configuration generation https://github.com/specs-feup/specs-lara https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P
Libversioning compiler	Sec. 5	https://github.com/specs-feup/specs-lara/tree/master/ANTAREX/PhaseSelectionAndOrdering/src
Split-compilation	Sec. 5	https://github.com/specs-feup/specs-lara https://antarex.fe.up.pt/owncloud/s/cueXIJO8hHAZ21P

Note that Clava currently supports only Ubuntu and CentOS. If-memo requires Linux.

7 References

1. Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In Proc. ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pages 147-162, New York, NY, USA, 2012. ACM.
2. Suresh Purini and Lakshya Jain. Finding good optimization sequences covering program space. ACM Transactions on Architecture and Code Optimization (TACO), 9(4):56:1-56:23, January 2013.
3. Luiz G.A. Martins, Ricardo Nobre, Alexandre C.B. Delbem, Eduardo Marques, and João M.P. Cardoso. Clustering-based selection for the exploration of compiler optimization sequences. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):8:1-8:28, March 2016.
4. Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration. In Proc. of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES '15). ACM, New York, NY, USA, 58-67.
5. Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. In Proc. 17th ACM Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, LCTES 2016, pages 21-30, New York, NY, USA, 2016. ACM.
6. Ricardo Nobre, Luís Reis, and João M. P. Cardoso. Compiler phase ordering as an orthogonal approach for reducing energy consumption. In Proc. of the 19th Workshop on Compilers for Parallel Computing, CPC '16.
7. Amir H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, J. Cavazos, MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning, TACO 14 (3), September 2017
8. Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, Cristina Silvano: COBAYN: Compiler Autotuning Framework Using Bayesian Networks. TACO 13(2): 21:1-21:25 (2016)