H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

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

Deliverable D2.5: DSL Support for Extra-Functional Characteristics

European Commission

Horizon 2020 European Union funding for Research & Innovation

Deliverable Title:	DSL Support for Extra-Functional Characteristics
Lead beneficiary:	Politecnico di Milano, Italy
Keywords:	Extra-Functional Properties, DSL
Author(s):	Giovanni Agosta, Andrea Bartolini, Loïc Besnard, João Bispo, João M. P. Cardoso, Stefano Cherubin, Davide Gadioli, Martin Golasowski, Imane Lasri, Gianluca Palermo, Pedro Pinto, Erven Rohou, Emanuele Vitali
Reviewer(s):	Federico Ficarelli (CINECA)
WP: **Nature:**	WP2 OTHER
Identifier:	D2.5
Delivery due date:	31/05/2018

Executive Summary	This document serves as a guide and accompanying report for the software distribution that composes Deliverable 2.5 of the ANTAREX project. The software distribution, which can be found at https://github.com/specs-feup/specs-lara/tree/master/2018%20DSD, provides a set of methods to embed specific technologies developed in ANTAREX within an HPC application, by means of the LARA DSL and its toolchains. D2.4 contains the full descrip-tion of the toolchain itself, which is therefore only briefly discussed here. The contents of the deliverable are organized as follows: Section 1 provides a summary of the ANTAREX approach to autotuning and optimization of HPC applications; Section 2 provides a discussion of the extra-functional properties targeted by the ANTAREX tool flow; Section 3 provides an in-depth analysis of the technology portfolio, with clear application examples for each technology; Section 4 shows how the technology portfolio is being applied in the ANTAREX Use Cases to achieve the desired extra-functional properties (this is a preliminary discussion, full reports will be presented in D4.4 and D5.5); Section 5 provides a preliminary evaluation of the effectiveness of the DSL in expressing the extra-functional properties (a final assessment will be provided in D6.9). Section 6 draws some conclusions and highlights the remaining work to be carried out in WP4 and WP5. A preliminary version of this document has been accepted for publication at the 2018 Euromicro Conference on Digital System Design (DSD 2018).
Approved and Issued by the Project Coordinator	**Date:** 15/06/2018

Project Coordinator: Prof. Dr. Cristina Silvano– Politecnico di Milano, Italy
Phone: +39-02-2399-3692- **Fax:** +39-02-2399-3411

Contents

1 Introduction

1.1 The ANTAREX Approach

The ANTAREX approach and related tool flow, as shown in Figure 1, operate both at design-time and runtime. The application functionality is expressed through C/C++ code (possibly including legacy code), whereas the extra-functional aspects of the application, including parallelisation, mapping, and adaptivity strategies, are expressed through DSL code (based on LARA) developed in the project. As a result, the expression of such aspects is fully decoupled from the functional code. The *Clava* tool is the centerpoint of the compile-time phase, performing a refactoring of the application code based on the LARA aspects, and instrumenting it with the necessary calls to other components of the tool flow.

The ANTAREX compilation flow leverages a runtime phase with compilation steps, through the use of partial dynamic compilation techniques enabled by *libVC*. The application autotuning, performed via the *mARGOt* tool, is delayed to the runtime phase, where the software knobs (application parameters, code transformations and code variants) are configured according to the runtime information coming from application self-monitoring as well as from system monitoring performed by the *ExaMon* tool. Finally the runtime power manager, *PowerCapper* is used to control the resource usage for the underlying computing infrastructure given the changing conditions. At runtime, the application control code, thanks to the design-time phase, now contains also runtime monitoring and adaptivity strategy code derived from the DSL extra-functional specification. Thus, the application is continuously monitored to guarantee the required Service Level Agreement (SLA), while communication with the runtime resource-manager takes place to control the amount of processing resources needed by the application. The application monitoring and autotuning is supported by a runtime layer implementing an application level collect-analyse-decide-act loop.

1.2 The ANTAREX DSL

HPC applications might profit from adapting to operational and situational conditions, such as changes in contextual information (e.g., workloads), in requirements (e.g., deadlines, energy), and in availability of resources (e.g., connectivity, number of processor nodes available). A simplistic approach to both adaptation specification and implementation (see, e.g., [1]) employs hard coding of, e.g., conditional expressions and parameterizations. In our approach, the specification of runtime adaptability strategies relies on a DSL implementing key concepts from Aspect-Oriented Programming (AOP) [2].

Our approach is based on the idea that certain application/system requirements (e.g., target-dependent optimizations, adaptivity behavior and concerns) should be specified separately from the source code that defines the functionality of the program. Those requirements are expressed as DSL aspects that embody strategies. An extra compilation step, performed by a *weaver*, merges the original source code and the aspects into the intended program [3]. Using aspects to separate concerns from the core objective of the program can result in cleaner programs and increased productivity (e.g., higher reusability of strategies). As the development process of HPC applications typically involves two types of experts (application-domain experts and HPC system architects) that can split their responsibilities along the boundary of functional description and extra-functional aspects, our DSL-aided toolflow provides a

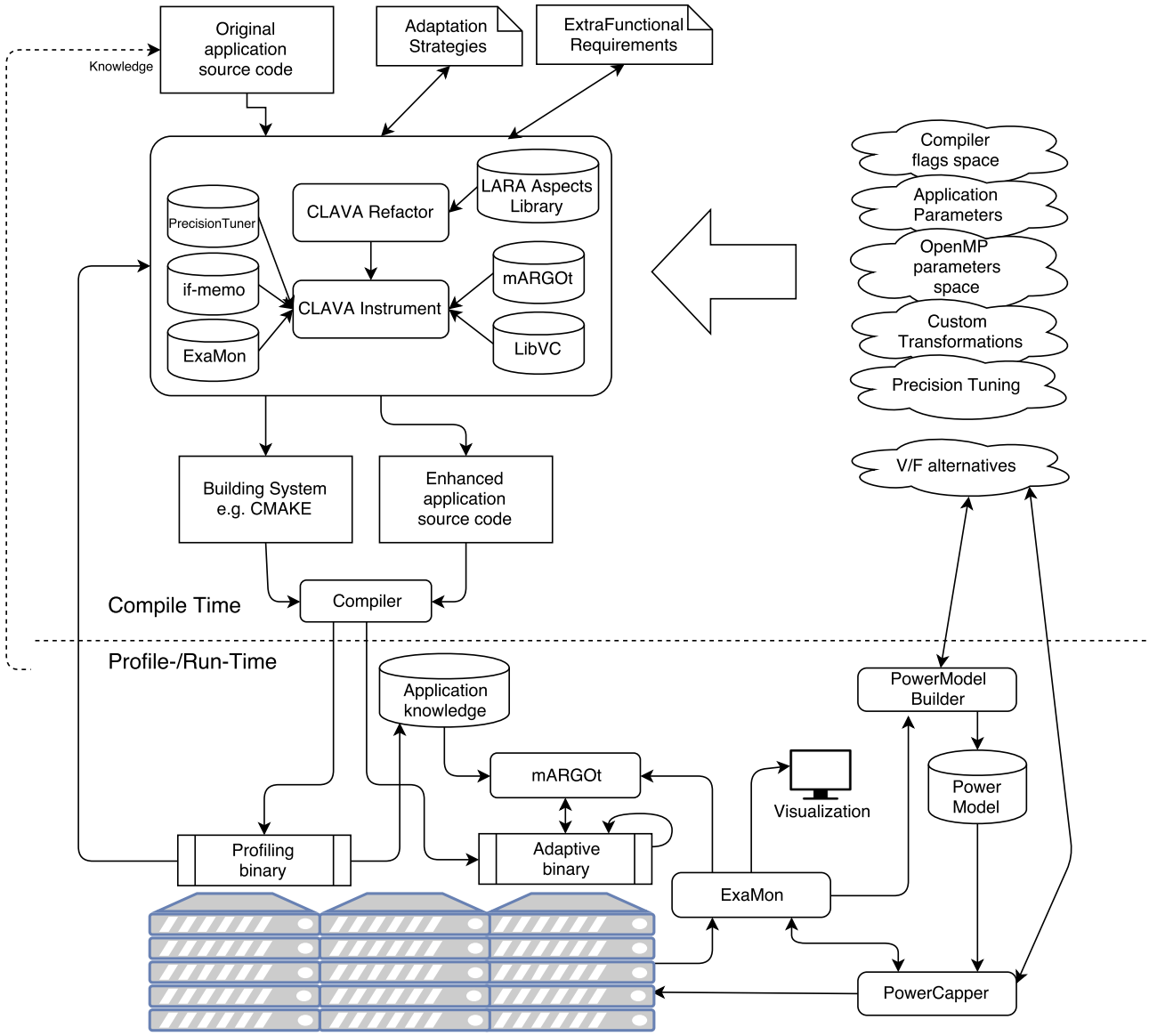

Figure 1: The ANTAREX Tool Flow

suitable approach for dealing and helping to express their concerns.

The ANTAREX DSL relies on the already existing DSL technology LARA [4, 5]. In particular, the LARA technology provides a framework that we adopted to implement the ANTAREX aspects and APIs. Moreover, we developed other LARA-related tools such as the *Clava*[1] weaver to leverage the rest of the ANTAREX tool flow.

LARA is a programming language that allows developers to capture non-functional requirements and concerns in the form of strategies, which are decoupled from the functional description of the application. Compared to other approaches that usually focus on code injection (e.g., [6]), LARA provides access to other types of actions, e.g., code refactoring, compiler optimizations, and inclusion of additional information, all of which can guide compilers to generate more efficient implementations. Additional types of actions may be defined in the language specification and associated weaver, such as software/hardware partitioning [7] or compiler optimization sequences [8]. One important feature of the LARA-aided source-to-source compiler developed in ANTAREX is the capability to refactor the code of the application in order to expose adaptivity behavior and/or adaptivity design points that can be explored by the ANTAREX autotuning component. In the following sections we show illustrative examples[2] of some of the strategies that can be specified using LARA in the context of a source-to-source compiler and currently used for one of the use cases.

The rest of this deliverable is organized as follows. In Section 2 we review the set of extra-functional properties targeted by the ANTAREX tool flow, and provide motivational scenarios to drive the definition of specific strategies. In Section 3 we review the technology portfolio provided by the ANTAREX tool flow. In Section 4, we show how the technology portfolio can be employed to target the specific extra-functional goals identified in Section 2 In Section 5 we provide an assessment of the impact of the proposed DSL on application specifications, while Section 4.2 gives an overview of how the Tool Flow has been used in one of the use cases. Finally, in Section 6 we draw some conclusions.

2 Extra-Functional Properties in ANTAREX

2.1 Performance

Since system performance became limited by the consumed or dissipated power, the optimization goal shifted toward efficiency in a wide range of scenarios, including HPC. An implication of this trend is that execution time is no more the only extra-functional concern, but it is possible to enhance an application via software-knobs that expose a trade-off between several extra-functional properties of the application itself. While the source code of an application define the procedure used to compute the desired result, extra-functional properties, such as time-to-solution or accuracy, depends also on the underlying architecture, on the system workload and on the actual input. For example, to further improve efficiency it is possible to produce a *good enough* result for the end-user by sparing the unnecessary computation effort. Therefore, application requirements are becoming more complex and

[1] https://github.com/specs-feup/clava

[2] Complete working versions for all examples can be found in https://github.com/specs-feup/specs-lara/tree/master/2018%20DSD

they target several extra-functional properties that are in conflict between each other, such as execution time, power consumption, and result accuracy.

To improve the system efficiency from a software point of view, the ANTAREX tool-chain exploits the DSL language to apply two types of strategies: the enhancement of the application via software-knobs, and the expression of extra-functional concerns as defined by mARGOt. A large class of applications directly expose software-knobs that alter its extra-functional behavior – for example the number of trials in a Monte Carlo approach. However, we might enhance the application by applying source-to-source transformations to expose additional software-knobs, which drive the techniques described in Section 3. Once the software-knobs have been defined, the ANTAREX tool-chain integrates mARGOt, which is the software component in charge of seizing optimization opportunities at run-time.

The ANTAREX tool-chain leverage the flexibility of LARA and mARGOt to accommodate different scenarios via the same blueprint: by tailoring the adaptation policies on the target application. For example, in the geometrical docking application, the ANTAREX tool-chain provides to end-user the possibility to control time-to-solution accuracy trade-off, using features of the input set. In the probabilistic time-dependent routing application, the ANTAREX tool-chain provides to end-user the possibility to control the response time according to features of the target path and to different types of users.

2.2 Power

Power consumption is raising of interest in HPC systems. Indeed as effect of Dennard's scaling in HPC computing systems are power limited. This happens at different granularities. At machine/system level the datacenter is constrained by the peak power consumption, and by fluctuations on the power consumption. Indeed at design time the facility is designed to host a given maximum power envelop for the expected peak power consumption of the system to be hosted. However during the life time of the facility several conditions may happen which may cause an higher demand of the power consumption higher than the one for which the facility has been designed. This may happen: (i) during transitioning periods between two systems, when the old one has to be dismissed and a new one to be installed there could be a period in which both the two systems are active but the sum of the two peak power consumptions would overpass the facility designed power envelop; (ii) if funding allows the procurement of a larger system than the one planned when the facility was designed, due to the fact that in average a supercomputer consumes the 70/80% of the peak power consumption, a machine with larger peak power consumption than the feasible one could be installed in the datacenter; (iii) as effect of natural disaster and seasonal behaviors a datacenter may be asked to operate its computing resources in a reduced power envelop; (iv) in smart grid power distribution a datacenter may be asked to modulate its power consumption to fullfill its power consumption requests. In all of these conditions the power of the supercomputer needs to be controlled and capped to a feasible value. Power capping strategies aim to do so by modulating the workload enetering the system as well as the performance of the computing nodes and elements.

In addition, the single computing element is power limited, indeed as effect of the end of Dennard's scaling the power needed to operate a computing element at its maximum speed is higher than the maximum one that can be provisioned throught the chip pins and to the one that can be dissipated throught standard cooling technologies. For this reason processing elements have built in strategies to limit the maximum power consumption that can be consumed.

In this power constrained scenario there is the need to operate the computing system and its resources at a reduced power envelop than the nominal one.

2.3 Energy

Not only power consumption is of interest of HPC systems, but also the associated energy consumption. The energy consumption of a supercomputer impacts significantly its total cost of ownership (TCO). Thus increasing the energy efficiency of its operation can increase the overall cost effectiveness and competitiveness of the entire datacenter. However the TCO is still dominated in today installation by the depreciation of the procurement costs of the computing resources (IT). It results that strategies that increases the energy efficiency by degrading the performance can detriment the overall economics of the datacenter as well as the fairness with the users. Today's two scenarios are of interests with respect to the increase of energy-efficiency: (i) promoting the increase of the energy-efficiency of codes running in the machine by letting the users be accounted based not only on the usage time of the machine, but also on the energy consumed by their jobs; (ii) Deliver run-time strategies to reduce the energy wasted in application slacks. In this perspective there are emerging tools to fine-tune the power consumption of computing resources reacting to short application phases in the applications. This with the goal of controlling the application slow down while saving power and energy. However the application needs to be linked and instrumented with specific libraries and directive to enable these service. Strategies to promote the user in inserting these directives as well as approaches to automate them have the potential to increase the datacenter cost effectiveness and carbon footprint.

3 ANTAREX Technology Portfolio

3.1 Precision Tuning

Error-tolerating applications are increasingly common in the emerging field of real-time HPC. Thus, recent works investigated the use of customized precision in HPC as a way to provide a breakthrough in power and performance. But, a user does not often think about the parametrization of its application in terms of types. So, in ANTAREX project, we develop a set of LARA aspects enabling the parametrization of the types of an application.

Figure 2 presents a part of a LARA strategy that changes all declarations of a certain type to a target type for an application. It allows to easily parametrize the application by, for example, replacing the `double` type by an abstract one (`\_DATATYPE\_`). Then, it becomes easy to produce different versions of the application by changing the definition of the abstract type (`\_DATATYPE\_`).

In particular, the LARA function `getNewType(RefType, oType, nType)` (used at line 31) returns the `oType` type if the type `RefType` does not reference the `oType` type or the type in which all the references to `oType` are replaced by `nType`.

Note that in the LARA aspects:

```
1
2 /** It  visits the application (variables, prototypes, typedefs, class, ...)
3     to substitute a type (oldType) by another one (newType). */
4 aspectdef rewType
5  input
6  oldType,
7  newType
8 end
9  call change_TypeVars( oldType, newType);
10  call change_TypePrototypes(oldType, newType);
11  call change_Typedefs(oldType, newType);
12  call change_TypeClass(oldType, newType);
13  call change_Casts(oldType, newType);
14  call change_NewExpr(oldType, newType);
15 end
16
17
18
19 /** Replace oldType by newType in the variable declarations,
20     including the parameters of the functions. */
21 aspectdef change_TypeVars
22 input
23   oldType,
24   newType
25 end
26
27 select vardecl end
28 apply
29   checkDefined($vardecl);
30   var vardeclType = $vardecl.type;
31   $ntype = getNewType(vardeclType, oldType, newType);
32   $vardecl.setType($ntype);
33 end
34 end
```

Figure 2: Example of LARA aspects to change the types of an application.

- all the specific names related to specific types must be renamed (e.g., `sqrtf` vs `sqrt` in `math.h`, the numeric limits C++ predefined values).
- the formats used in the I/O operations in C must be analyzed and isolated to support a new format.
- a mecanism has been introduced to preserve some types during the transformations: for example, the type `DOUBLE_TYPE` with the following definition `typedef double DOUBLE_TYPE` may be declared 'unchangeable' for a translation `double` to `float`.

The parametrization of this changes is extracted in a specific file. For known types (`double`, `float`), these parameters are generated automatically. But, for new representations (e.g. fixed-point, half-precision floating-point) they must be provided by the user.

This mechanism is illustrated on figures 3, 4 and 5: the figure 3 represents a part of a original method that is automaticaly translated by LARA in the code of the figure 4 (the weaved code) and the parametrization of the application is generated in a file shown on figure 5.

Such a mechanism allows the user to quickly test the effect of precision on its application. It has been applied to the use cases with success (`double` to `float` transformation).

```
1
2 BetweennessResult Betweenness::Calculate(int startVertex, int endVertex)
3 {
4   cout << "start " << startVertex << ", end " << endVertex << endl;
5   int vertices = graph->GetVertices();
6   int edges = graph->GetEdges();
7   double *betweenness = new double[vertices];
8   double *edgeBetweenness = new double[edges];
9   for (size_t i = 0; i < vertices; i++)
10    {
11      betweenness[i] = 0;
12    }
13  for (size_t i = 0; i < edges; i++)
14    {
15      edgeBetweenness[i] = 0;
16    }
17
18  priority_queue<KeyValuePair, vector<KeyValuePair>, greater<KeyValuePair>> Q;
19  stack<int> S;
20  double *dist = new double[vertices];
21  list<int> *Pred;  //list of predecessors is faster
22  double *sp = new double[vertices];
23  double *delta = new double[vertices];
24  bool *isInStack = new bool[vertices];
25
26  DOUBLE_TYPE shortestPathsWeightSumMetric = 0;
27
28  for (int s = startVertex; s < endVertex; s++)
29    {
30 ...
31    }
```

Figure 3: Custom precision: a part of the original C++ code.

3.2 Code Versioning

One of the strategies supported by the ANTAREX toolflow is the capability to generate different versions of the same function and select the one that satisfies certain requirements at runtime. Figure 6 shows an aspect that clones a set of functions and changes the types of the newly generated clones. Each clone has the same name as the original with the addition of a provided suffix. We start with a single user-defined function which is cloned by the aspect `CloneFunction` (called in line 13). Then, it recursively traverses calls to other functions inside the clone and generates a clone for each of them. Inside the clones, calls to the original functions are changed to calls to the clones instead, building a new call tree with the generated clones. At the end of the aspect `CreateFloatVersion` (lines 16–17,) we use the previously defined `ChangePrecision` aspect to change the types of all generate clones.

The aspect `Multiversion` – in Figure 7 – adapts the source code of the application in order to call the original version of a function or a generated cloned version with a different type, according to the value of a parameter given by the autotuner at runtime. The main aspect calls the previously shown aspect, `CreateFloatVersion`, which clones the target function and every other function it uses, while also changing their variable types from `double` to `float` (using the aspects presented in Figure 6 and Figure 2). This is performed in lines 8–9 of the example. From lines 13 to 34, the `Multiversion` aspect generates and inserts code in the application that is used as switching mechanism between the two versions. It starts by declaring a variable to be used as a knob by the autotuner, then it generates the code for a switch statement and replaces the statement containing the original call

```
}

BetweennessResult Betweenness::Calculate(int startVertex, int endVertex) {
   cout << "start " << startVertex << ", end " << endVertex << endl;
   int vertices = this->graph->GetVertices();
   int edges = this->graph->GetEdges();
   _DATATYPE_ * betweenness = new _DATATYPE_[vertices];
   _DATATYPE_ * edgeBetweenness = new _DATATYPE_[edges];
   for(unsigned long i = 0; i < vertices; i++) {
      betweenness[i] = 0;
   }
   for(unsigned long i = 0; i < edges; i++) {
      edgeBetweenness[i] = 0;
   }
   priority_queue<KeyValuePair, vector<KeyValuePair>, greater<KeyValuePair>> Q;
   stack<int> S;
   _DATATYPE_ * dist = new _DATATYPE_[vertices];
   list<int> * Pred; //list of predecessors is faster
   _DATATYPE_ * sp = new _DATATYPE_[vertices];
   _DATATYPE_ * delta = new _DATATYPE_[vertices];
   bool * isInStack = new bool[vertices];
   DOUBLE_TYPE shortestPathsWeightSumMetric = 0;

   for(int s = startVertex; s < endVertex; s++) {
      //Single source shortest-paths problem
      {
...
      }
```

Figure 4: Custom precision: the weaved code produced for the code of 3

with the generated switch code. Finally, in lines 36–38, the aspect surrounds both calls (original and float version) with timing code. An excerpt of the resulting C code can bee seen in Figure 8.

In the ANTAREX toolflow, the capability of providing several versions of the same function is not limited to static features. LIBVERSIONINGCOMPILER [9] (abbreviated LIBVC) is an open-source C++ library designed to support the dynamic generation and versioning of multiple versions of the same compute kernel in a HPC scenario. It can be used to support continuous optimization, code specialization based on the input data or on workload changes, or to dynamically adjust the application, without the burden of a full just-in-time compiler. LIBVC allows a C/C++ compute kernel to be dynamically compiled multiple times while the program is running, so that different specialized versions of the code can be generated and invoked. Each specialized version can be versioned for later reuse. When the optimal parametrization of the compiler depends on the program workload, the ability to switch at runtime between different versions of the same code can provide significant benefits [10, 11]. While such versions can be generated statically in the general case, in HPC execution times can be so long that exhaustive profiling may not be feasible. LIBVC instead enables the exploration and tuning of the parameter space of the compiler at runtime.

Figure 9 shows an example of usage of LIBVC through LARA, which demonstrates how to specialize a function. The user provides this aspect with a target function call and a set of compilation options. These include compiler flags and possible compiler definitions, e.g., data discovered at runtime, which is used as a compile-time constant in the new version. Based on the target function call, the aspect finds the function definition which is passed to the library. After the options are set, the original function call is replaced with a call of the newly compiled and loaded specialized version of the kernel.

It is worth noting that the combination of LARA and LIBVC can also be used to support compiler flag selection and phase-ordering both statically and dynamically [12, 13].

```
#ifndef _INRIA_PRECISION_DEFS_H_
#define _INRIA_PRECISION_DEFS_H_

#ifdef _DOUBLE_
#define fabs_DATATYPE_ fabs
#else
#define fabs_DATATYPE_ fabsf
#endif

#ifdef _DOUBLE_
#define _DATATYPE_ double
#else
#define _DATATYPE_ float
#endif

#endif
```

Figure 5: Custom precision: an example of generated parameters.

```
import ChangePrecision;
import clava.ClavaJoinPoints;

aspectdef CreateFloatVersion
  input $func, suffix end
  output $clonedFunc end

  $double = ClavaJoinPoints.builtinType('double');
  $float = ClavaJoinPoints.builtinType('float');

  /* clone the target functions and the child calls */
  var clonedFuncs = {};
  call cloned : CloneFunction($func, suffix, clonedFuncs);

  /* change the precision of the cloned function */
  for($clonedFunc of clonedFuncs)
    call ChangePrecision($clonedFunc, $double, $float);

  $clonedFunc = cloned.$clonedFunc;
end
```

Figure 6: Example of LARA aspect to clone an existing function and change the type of the clone.

3.3 Memoization

Memoization is the technique of saving the results of computations so that future executions can be omitted when the same inputs repeat. In the deliverable 2.4, we have detailed the memoization support: the main goal of the library developed and its main features considering the upload of a table, the dynamic update policies, etc. and the link to the code of the application.

In this section, we present the memoization proposed in the ANTAREX project by relying on aspects programmed using the DSL. The advantage of these aspects is that the memoization is integrated into the application without requiring user modifications of the source code. The code generated by Clava is then compiled and linked with the associated generated memoization library.

We have shown that the generation of a memoization library starts from the definitions of a file (`funs-static.def`). So, the main goals of the LARA aspects developed is the generation of such definitions and the modification of the source code as shown in Figure 10.

An example of a LARA aspect for memoization is shown in Figure 11. It defines the memoization

```
1  import CreateFloatVersion;
2  import lara.code.Timer;
3  import clava.ClavaJoinPoints;
4
5  aspectdef Multiversion
6    input $func, knobName end
7
8    call fVersion : CreateFloatVersion($func, "_f");
9    var $floatFunc = fVersion.$clonedFunc;
10   var timer = new Timer();
11
12    /* Identify call by name... */
13    select function.body.stmt.call{$func.name} end
14    apply
15     /* ... and by type signature */
16     if(!$func.functionType.equals($call.functionType))
17       continue;
18
19     /* Add knob for choosing the version */
20     $int = ClavaJoinPoints.builtinType('int');
21     $body.exec addLocal(knobName, $int, 0);
22
23     /* create float declaration for first argument */
24     var $arg = createFloatArg($call.args[0]);
25     /* Create call based on float version of function */
26     $floatFunc.exec $fCall : newCall([$arg, $call.args[1]]);
27     /* Copy current call */
28     $call.exec $callCopy : copy();
29
30     /* Create switch */
31     var $condition = ClavaJoinPoints.exprLiteral(knobName);
32     var switchCases = {0: $callCopy, 1: $fCall};
33     call switchJp : CreateSwitch($condition, switchCases);
34     $stmt.exec replaceWith(switchJp.$switch);
35
36     /* Time calls to both original and float functions*/
37     timer.time($callCopy, "Original time:");
38     timer.time($fCall, "Float time:");
39    end
40 end
```

Figure 7: Example of LARA aspect that generates an alternative version of a function and inserts a mechanism in the code to switch between versions.

(lines 1-13) of a method (`aMethod`) of a class (`aClass`) with `nbArg` parameters (`nbArg` ≤ 3) of same type as the returned type (`Type`) restricted to `int, float, double`. Note that the inputs `nbArg` and `Type` are required to manage the overloading mechanism in languages (such that C++) that sport multiple argument dispatch. Other parameters (from line 7) are provided to improve several memoization approaches. The user can specify:

- the policy in case of conflicts regarding the same table entry (line 10): replacement or not in case of conflict to the same entry of the table for different parameters of the memoized function,
- the number of bits not to be considered in each input parameter (line 11) for double and float types. This technique is a kind of approximation: the result of the memoized function/method is the same for almost equal input arguments.
- a name of an input file (line 7) that will be used to initialize the internal table.
- a name of an output file (line 8) that will be used to save the internal table at the end of the execution. This result may be used as an input for a subsequent execution.

```
switch (version) {
  case 0: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_0);
    SumOfInternalDistances(atoms, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_0);
    double time_0 = calc_time(time_start_0, time_end_0);
    printf("Original time:%fms\n", time_0);
  }
  break;
  case 1: {
    clock_gettime(CLOCK_MONOTONIC, &time_start_1);
    SumOfInternalDistances_f(atoms_f, 1000);
    clock_gettime(CLOCK_MONOTONIC, &time_end_1);
    double time_1 = calc_time(time_start_1, time_end_1);
    printf("Float time::%fms\n", time_1);
  }
  break;
}
```

Figure 8: Excerpt of the C code resulting from the generation of alternative code versions.

- the size of the table (line 12).

After some verifications, not detailed here, on the parameters (lines 14-15), the method is searched (lines 17-24). Then, in case of success, the code of the wrapper is added (line 28) to produce the memoization library, and (line 30) the code of the application is modified to call the created “wrapper” that is also declared as a new method of the class.

Moreover, some variables are exposed for autotuning in the memoization library. For each function or method, a variable that manages the dynamical "stop/run" of the memoization is exposed, as well as the variable that manages the policy to use in case of conflict to the table.

A set of LARA aspects are integrated into Clava tool. We distinguish:

- the memoization of the mathematical functions (from `math.h\verb`) :
  - `Memoize\_AllMathFunctions()` used to memoize all the referenced mathematical memoizable functions of an application with default parameters.
  - `Memoize\_MathFunctions(['log', 'sin'])` used to memoize some mathematical functions with default parameters (for example log and sin functions).
  - `Memoize\_MathFunction('log')` used to memoize one mathematical function with default parameters (for example log function).
  - `Memoize\_MathFunction\_ARG('log', 0, 'inputFile', no, 'ouputfile', n` used to memoize one mathematical function with specific parameters as explained above.
- the memoization of C functions
  - `Memoize\_Function ('foo')` generates the required informations for the memoization library for a user C function (`foo`) with the internal default parameters.
  - `Memoize\_Function\_ARGS ('foo', 'none', 'no', 'none','no',0, 4096)` this aspect is similar to the previous one with specific user parameters.
- the memoization of C++ methods

```
import antarex.libvc.LibVC;

aspectdef SimpleLibVC

  input
    name, $target, options
  end

  var $function = $target.definition;
  var lvc = new LibVC($function, {logFile:"log.txt"}, name);

  var lvcOptions = new LibVCOptions();
  for (var o of options) {
    lvcOptions.addOptionLiteral(o.name, o.value, o.value);
  }
  lvc.setOptions(lvcOptions);

  lvc.setErrorStrategyExit();

  lvc.replaceCall($target);
end
```

Figure 9: Example of LARA aspect to replace a function call to a kernel with a call to a dynamically generated version of that kernel.

- Memoize_Method ('Test', 'foo') generates the required informations for the memoization library for a method of a class with the internal default parameters (here the Test::foo method)
- Memoize_Method_ARGS ('Test', 'foo', 'none', 'no', 'res.data', 'yes', 0, 2048) is similar to the previous one with user specific parameters.
- Memoize_Method_overloading('Test','foo','float', 2) is an aspect to fix overloading. It specifies the memoization of the Test::foo method with 2 arguments of same type (float) Memoize_Method_overloading_ARGS with default parameters.
- Memoize_Method_overloading_ARGS('Test','foo,'float', 2,'none', 'no', 'none', 'yes', 17, 2048) the most general for C++, similar to the previous one with specific user parameters.

The *Use Case 2: Self-Adaptive Navigation System* has been used to proof the concept of the memoization. In this application, the computation of a metric has been isolated in a method, this method has been then memoized. When the computation of the metric is simple the memoization does not produce any gain in term of time. The gain increases when the compution becomes more complex (a combination of mathematical functions).

To be complete about the memoization, a LARA aspect is also proposed to automatically detect the memoizable functions or methods. A function/method is declared memoizable iff:

- it satisfies the conditions of the memoization library about the interface: it must have less than or equal to 3 inputs arguments of the same type, and must returns a value of same type than the inputs arguments.
- it references only constants or local variables.
- and, all the functions/methods it references are memoizable.

The aspect returns the list of the detected memoizable objects to the user. It may decide to apply or not

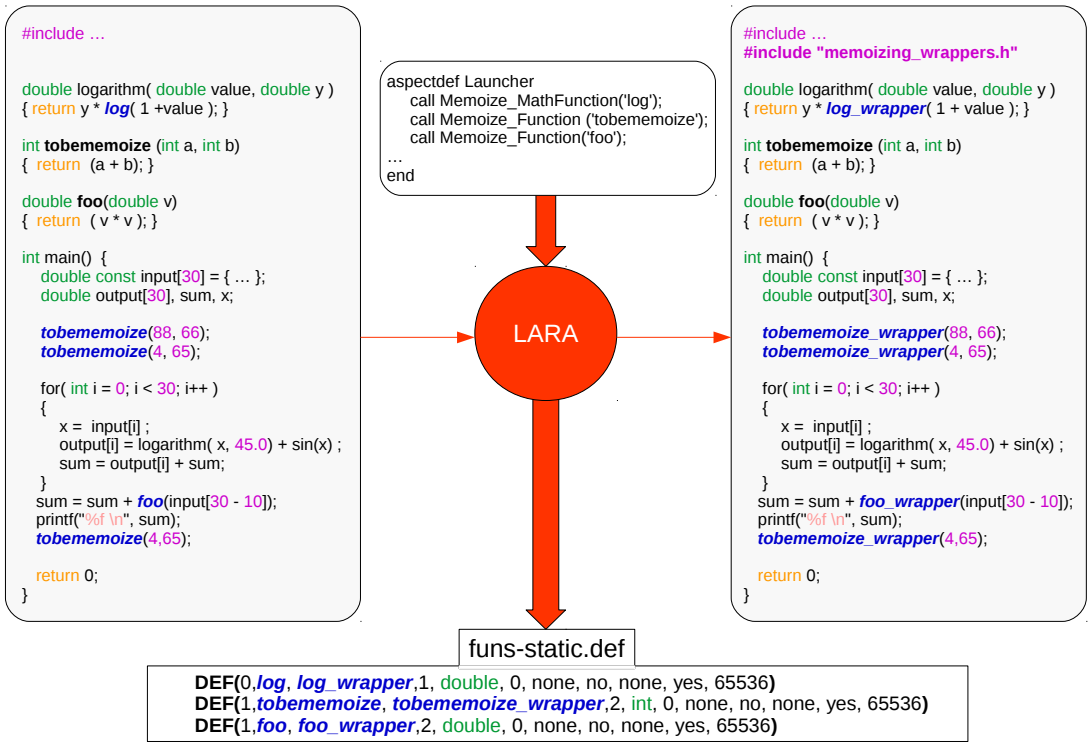

Figure 10: The memoization using LARA

the memoization on these returned elements.

3.4 Split compilation

Compilers rely on hundreds of optimizations to deliver performance. Optimization levels (such as `-O3`) are highly tuned to generate good code on a set of representative benchmarks. For a single application, however, the heuristics can often be significantly surpassed. Iterative compilation has been proposed in the late 1990's to explore a large domain space in order to find better optimization sequences. The drawback is the need for a large number of executions. We propose to take advantage of long running loops to explore the impact of several optimization sequences at once, thus reducing the number of necessary runs. We rely on a variant of loop peeling which splits a loop into into several loops, with the same body, but a subset of the iteration space. New loops execute consecutive chunks of the original loop. We then apply different optimization sequences on each loop independently. Timers around each chunk observe the performance of each fragment.

This technique may be generalized to combine compiler options and different implementations of a function called in a loop. It is useful when, for example, the profiling of the application shows that a function is critical in term of execution time. In this case, the user must try to find the best implementation of its algorithm.

The application of this methodology is illustrated in figure 12: consider 2 implementations of a function (`FOO`) and 2 sets of options. The loop (lines 4-5), calling the function `FOO`, is split into a sequence of

```
1 aspectdef Memoize_Method_overloading_ARGS
2 input
3   aClass,      // Name of a class
4   aMethod,     // Name of a method of the class aClass
5   pType,       // Name of the selected type
6   nbArgs,      // Number of parameters of the method
7   fileToLoad, // filename for init of the table, or 'none'
8   FullOffLine,// yes/no. yes for a fully offline strategy
9   FileToSave, // filemane to save the table, or 'none'
10  Replace,     // yes/no. yes: always replace in case of collisions
11  approx,      // Number of bits to delete for approximation.
12  tsize        // Size of the internal table.
13 end
14 // Control on the parameters of the aspect: nbArgs in [1,3]
15    ...
16 // Searching the method.
17 var MethodToMemoize, found=false;
18 select class{aClass}.method{aMethod} end
19 apply
20  if (! found) {
21    found = isTheSelectedMethod($method, nbArgs, pType);
22    if (found) MethodToMemoize=$method;
23  }
24 end
25 if (!found)
26  { /* message to the user */}
27 else {
28  GenCode_CPP_Memoization(aClass, aMethod, pType, nbArgs,
29  fileToLoad, FullOffLine, FileToSave, Replace, approx, tsize);
30  call CPP_UpdateCallMemoization(aClass, aMethod, pType, nbArgs);
31 }
32 end
```

Figure 11: An example of LARA aspect defined for the memoization.

blocks (lines 11-17, lines 20-26, lines 29-35, lines 38-44) with the same structure:

- the call to the load of a version of the function compiled with a set of compiler options,
- a loop with the same body as the original one, with a number of iterations equal to (N/4), where N is the number of iterations of the original loop. Timers are added around such a loop to get its execution time.

Then, after a step of "learning", the decision must be taken to select the best version..

One can observe that this methodology may be implemented in LARA using the cloning of code and the code versioning 3.2.

To prove the concept, this methodology has been applied to the Use Case 1 (Computer Accelerated Drug Discovery, version 3) of ANTAREX. It produced an important gain in term of execution time (around 24%).

3.5 Self-Adaptivity & Autotuning

In ANTAREX, we consider each application's function as a parametric function that elaborates input data to produce an output (i.e., $o = f(i, k_1, \ldots, k_n)$), with associated extra-functional requirements. In this context, the parameters of the function $(k_1, \ldots, k_n)$ are software-knobs that modify the behavior of the application (e.g., parallelism level or the number of trials in a MonteCarlo simulation).

The main goal of mARGOt [3] [14] is to enhance an application with an adaptive layer, aiming at tuning the software knobs to satisfy the application requirements at runtime. To achieve this goal, the mARGOt dynamic autotuning framework developed in ANTAREX is based on the MAPE-K feedback loop [15]. In particular, it relies on an application knowledge, derived either at deploy time or at runtime, that states the expected behavior of the extra-functional properties of interest. To adapt, on one hand mARGOt uses runtime observations as feedback information for reacting to the evolution of the execution context. On the other hand, it considers features of the actual input to adapt in a more proactive fashion. Moreover, the framework is designed to be flexible, defining the application requirements as a multi-objective constrained optimization problem that might change at runtime.

To hide the complexity of the application enhancement, we use LARA aspects for configuring mARGOt and for instrumenting the code with related API. Figure 13 provides a simple example of a LARA aspect where mARGOt has been configured (lines 5-20) to actuate on a software knob *Knob1* and target *error* and *throughput* metrics [16]. In particular, the optimization problem has been defined as the maximization of the *throughput* while keeping the *error* under a certain threshold. The last part of the aspect (lines 23-27) is devoted to the actual code enhancement including the needed mARGOt call for initializing the framework and for updating the application configuration. The declarative nature of the LARA library developed for integrating mARGOt simplifies its usage hiding all the details of the framework.

3.6 Monitoring

Today's processing elements embed the capability of monitoring their current performance efficiency by inspecting the utilization of the micro-architectural components as well as a set of physical parameters (i.e., power consumption, temperature, etc). These metrics are accessible through hardware performance counters which in x86 systems can be read by privileged users, thus creating practical problems for user-space libraries to access them. Moreover, in addition to sensors which can be read directly from the software running on the core itself, supercomputing machines embed sensors external to the computing elements but relevant to the overall energy-efficiency. These elements include the node and rack cooling components as well as environmental parameters such as the room and ambient temperature. In ANTAREX, we developed ExaMon[17] (*Exa*scale *Mon*itoring) to virtualise the performance and power monitoring access in a distributed environment. ExaMon decouples the sensor readings from the sensor value usage. Indeed, ExaMon uses a scalable approach were each sensor is associated to a sensing agent which periodically collects the metrics and sends the measured values with a synchronized time-stamp to an external data broker. The data broker organises the incoming data in communication channels with an associated topic. Every new message on a specific topic is then broadcast to the related subscribers, according to a list kept by the broker. The subscriber registers a callback function to the given topic which is then called every time a new message is received. To let LARA take advantage of this monitoring mechanism we have designed the Collector API, which allows the initialization of the Collector component associated with a specific topic that keeps an internal state of the remote sensor updated. This internal state can then be queried asynchronously by the Collector API to gather its value. LARA aspects have been designed to embed the Collector API and to make the application code self-aware.

Figure 14 shows a usage example of ExaMon through LARA, which subscribes to a topic on a given

[3] https://gitlab.com/margot_project/core

broker and inserts a logging message in the application. To define the connection information, the user has to provide the address to connect to, as well as the name of the topic to subscribe. As for the integration in the original application code the user has to provide a target function, where the collector will be managed, and a target statement, where the query of the data and logging will be performed.

3.7 Power Capping

Today's computing elements and nodes are power limited. For this reason state-of-the-art processing elements embed the capability of fine-tuning their performance to control dynamically their power consumption. This includes dynamic scaling of voltage and frequency, and power gating for the main architectural blocks of the processing elements, but also some feedback control logic to keep the total power consumption of the processing element within a safe power budget. This logic in x86 systems is named RAPL [18]. Demanding the power control of the processing element entirely to RAPL may not be the best choice. Indeed it has been recently discovered that RAPL is application agnostic and thus tends to waist power in application phases which exhibit IOs or memory slacks. Under these circumstances there are operating points that proved to be more energy efficient than the ones selected by RAPL while still respecting the same power budget [19]. However these are only viable if the power capping logic is aware of the application requirements. To do so, we have developed a new power capping run-time based on a set of user space APIs which can be used to define a relative priority for the given task currently in execution on a given core. Thanks to this priority, the run-time is capable of allocating more power to the higher priority process [20, 21]. In ANTAREX, these APIs can be inserted by LARA aspects in the application code.

4 Strategies

This section describes the experiments, manual and with Clava support, that were performed on both uses cases.

4.1 Use Case 1

We performed manual experiments on several versions of the UC1 application, as it was being developed. This included code transformations and data structure rearrangements on critical parts of the applications, hotspots where the application spent most of its execution time.

The main idea behind this work was to understand which code transformation strategies have an impact on the performance of the UC1 application, both in terms of execution time and energy consumption, and measure this impact. Although we are interested in both metrics, the results collected mainly focus on execution time.

Our intention afterwards is to implement some of these transformations in Clava, either as actions, aspects or libraries the developer can use. The choice of which transformations to include will be made considering their benefits and how feasible the implementation is. For instance, some of the

transformations can be translated into small actions that aim at improving a well-defined region of the code, such as a function. These include:

- Avoiding passing arguments by copy, using references when possible (may require some user input);
- Avoiding unnecessary casts;
- Replacing small portions of code with computationally cheaper, but equivalent, code.

These were all used in the exploration and development of a more performant version of the UC1 code. We can see how these transformations translate well to Clava support, either through actions or aspects. On the other hand, the transformation that is responsible for the largest improvement is likely not as easily translated to LARA. This consisted in using a different data structure to represent the molecule's atoms positions during computation inside the application hotspot to improve spatial data locality. More specifically, storing subsequent X, Y and Z coordinates of each atom in three separated data structures where subsequent atom positions over a given coordinate are represented in subsequent memory positions. Figure 15 explains how the transformation works conceptually. The array of atoms (before), each with its coordinates, is transformed into three arrays (after), one for each coordinate.

Execution time improved by a factor of $9\times$ when executing the complete UC1 application on the same hardware, while the execution time of the hotspot improved by a factor of $10.7\times$.

4.1.1 Experiments Using Accelerators

Since the application sports a compute-intensive, data-parallel hotspot, one can say that accelerators such as the Xeon Phi and GPUs are suitable to improve its performance, provided the hotspot code is efficiently mapped to the target accelerator of choice. However, between the Xeon Phi and a contemporary high-end GPU, the latter is capable of a much higher throughput in compute-intensive, data-parallel code. For instance, add-in cards with the AMD Vega 10 XT or the NVIDIA GP100 GPUs are each capable of more than 10 TFLOPS in single-precision floating point, while the Xeon Phi 7210P (the model in the ANTAREX workstation) is capable of only 650.5 GFLOPS (also in single-precision).

Preliminary tests of the UC1 application on the Xeon Phi and on a GPU confirmed our assumption that, given the nature of the hotspot code, a GPU is better suited regarding acceleration potential. Energy-to-solution was not measured in the GPU versions, but given that TDP measures announced by Intel for the Xeon Phi 7120P (300W) and for typical high-end GPUs (250 300W) is very comparable, it is to expect that GPUs will be much more efficient for executing code such as this application's hotspot.

To use the GPU efficiently in UC1 required porting not only the code that computes the molecular docking score, but also the code that rotates and translates the molecules. Otherwise, given the data-parallel nature of the rotations and translation, performing the rotations on the CPU makes the application not only slower in terms of computation, but also much slower overall, because of having to transfer a much larger quantity of data to the GPU if rotating on the host-side. Hence, several functions had to be ported to GPU code (inlined in a single GPU kernel). We developed the GPU-accelerated version building on top of the optimized CPU version. This was beneficial because when starting to develop the GPU-accelerated version, we had already modified the hotspot code to improve data locality, which simplifies the code for copying the molecule's atoms coordinates from the host to the accelerator.

Figure 16 shows the performance gain obtained with the optimized CPU version, and when targeting a Vega 10 XT GPU with an OpenCL implementation that offloads the application's hotspot code to the GPU. In both cases, the baseline is the CPU-only version before optimization.

The hotspot code is much faster on the GPU, resulting in a $65.1\times$ speedup (the CPU-only optimized version results in a speedup of $10.7\times$). If using single-precision instead, the speedup versus the original version becomes $124.6\times$. However, when considering the execution of the whole application, the GPU-accelerated version is $28.6\times$ faster if using double-precision floating-point, and $35.2\times$ faster if using single-precision; revealing that, at this point in the GPU accelerated versions, what was previously the application hotspot represents now an even smaller percentage of the execution time. Moreover, the single-precision version on the GPU did not result in a significant loss of precision for most evaluated pockets because the rotations are performed over the same initial atom positions (i.e., each GPU kernel instance performs the full X, Y rotation starting from the initial atom coordinates).

For the implementation of OpenCL version of the application, Clava offers some support that can be leveraged by developers. There is a library, `KernelReplacer`, that can be imported into a user aspect and used to replace a target call with an OpenCL kernel invocation. In this case, after manually writing the kernels, a developer can use the library to generate all the necessary OpenCL code (e.g., context, device and buffer management) and insert it in the application, replacing the original call. The developer needs to provide some information, such as the local sizes and number of iterations of the problem space, as well as a file with the kernel code. For more information on this library, please see the report accompanying deliverable 2.4.

4.2 Use Case 2: Self-Adaptive Navigation System

In this section, we provide an overview of the application of the ANTAREX tool flow to the Self-Adaptive Navigation System developed in Use Case 2. The system is designed to process large volumes of data for the global view computation and to handle dynamic loads represented by incoming routing requests from users of the system. Both disciplines require HPC infrastructure in order to operate efficiently while maintaining contracted SLA. Integration of the ANTAREX self-adaptive holistic approach can help the system to meet the mentioned requirements and pave the way to scaling its operation to future Exascale systems.

Core of the system is a routing pipeline with several stages which uses our custom algorithm library written in C++. The library provides an API for the individual routing algorithms and for data access layer, which provides abstraction of a graph representation of the road network. The graph is stored in a HDF5 file, which is a well known and convenient storage format for structured data on HPC clusters.

As an example, we are using LARA aspects to generate C++ code for mapping native data types to types defined by the HDF5 API. The aspects are applied using the *Clava* tool which is a C++ frontend for the LARA toolchain. The *Clava* tool is integrated in our CMake-based build process as a custom build step, which parses the C++ structures representing the routing graph in memory and produces part of the HDF5 data access API. Details of the implementation can be found in [22]. Using the same process, other LARA aspects can be easily applied on the source code of the library, which greatly simplifies integration of other tools of the ANTAREX toolchain, such as mARGOt [14].

Furthermore, the mARGOt [14] autotuner is used in the Probabilistic Time-Dependent routing (PTDR) algorithm [23] to dynamically adjust the number of Monte Carlo samples used for the particular routing

request. This parameter directly affects load generated by the PTDR stage and precision of its output. The autotuner uses operation point lists generated by a Design Space Exploration phase. The operation points in the context of PTDR are represented by a number of MC samples as an adjustable algorithm parameter and expected values of various metrics. The autotuner then dynamically selects the operation point according to the current request input. This approach can significantly reduce computational load generated by the PTDR phase, contributing to the overall efficient operation of the system.

Currently, our codebase is ready to use the DSL to integrate other tools from the ANTAREX tool flow. The autotuner is manually integrated in the routing pipeline, while verification of its correct operation is ongoing. The next step is to use LARA to integrate the autotuner to the target application and evaluate its impact.

We have developed a server-side routing dashboard web application which is used to monitor the current status of the routing service. The application also provides a consistent environment for testing the service performance. It provides a way to execute a benchmark of the service by adjusting its parameters and sending a pre-defined set of routing requests. The service performance is then measured and results of the testing are stored for further analysis. The application also provides a consistent visualisation of the results which can be used for further analysis. This infrastructure will be used for validation of the ANTAREX tools integrated in the routing service [24].

4.2.1 Autotuning Experiments

We used Clava and one of its libraries, the LARA Autotuning Tool (LAT)[4], to perform an automatic exploration of the Monte Carlo application of UC2. This tool is able to automatically explore the design space of an application and requires the user to specify the adaptation scope, the measuring point, which metrics to measure and the ranges of the variables to explore. For more information on the LAT, please see the report accompanying deliverable 2.4.

In this work we focused on the number of threads used by the OpenMP version of the application and tried the range $[1, 32]$. Since the exploration of OpenMP applications and their parameters was one of the key concerns of the development of LAT, it is easy to express this design-space exploration using this library. Although we focused only on the number of threads, this analysis could easily be extended to consider scheduling policies and chunk sizes.

In Figure 17 we can see, for each tested number of threads, the execution time and energy consumption of the Monte Carlo application, normalized against the results of using a single thread (with OpenMP). For instance, increasing the number of threads from 1 to 2 results in the application taking around 53% of time to execute and needing 63% of the energy. This application is able to scale well with an increasing number of threads, so the results are not at all surprising and it's easy to conclude that, if able, one should use as many threads as possible. It's still interesting to note that execution time decreases faster than energy consumption as the number of threads increases. The results show diminishing returns on the achieved savings as the number of used threads grows. After a certain point, application quality metrics regarding execution time and energy consumption may be met and it becomes unnecessary to use more resources, which may possibly be used by other applications running on the node.

This exploration illustrates how well-suited LAT is to perform this kind of design-space exploration and tunning. This is especially true for OpenMP application parameters but can also be applied to

[4] https://github.com/specs-feup/LAT-Lara-Autotuning-Tool

arbitrary variables in the application code. These can represent different algorithms to perform some computation and their parameters.

4.2.2 Memoization Experiments

We performed some memoization experiments on the UC2 application code, namely on the betweenness application. This was performed as an exploration to assess the potential of the memoization techniques (and their implementation) on an application such as betweenness. We first changed how a metric, sum of edge weights in the shortest path, is calculated. Originally, as the name indicates, this is calculated simply by summing the weights of every edge in the found shortest path. We changed this metric calculation to an artificial metric that performs more work and uses functions from the math library, which are common targets for memoization. The new metric remains the same but the weight of each edge is now given by:

$$weight_{new} = \left(weight_{old} + \frac{\ln\left(weight_{old}\right)}{weight_{old}} + \frac{weight_{old}}{\sin\left(weight_{old}\right)} \right) \times weight_{old}$$

We applied our memoization techniques to this new version of the application. Most of the effort here was performed automatically, since we used the memoization library shipped with Clava. We tested the application in two different scenarios, using data from two different cities, Brno and Vienna. Compared to executions of the original version of the application, we achieved speedups of 12.83% for Brno and 13.64% for Vienna. For additional details on the memoization support provided by Clava through this library, please see the report accompanying deliverable 2.4.

5 Preliminary Evaluation

Tables 1 and 2 show static and dynamic metrics collected for the weaving process of the presented examples. In Table 1, we can see the number of logical lines of source code for the LARA strategies, as well as for the input code and generated output code (the SLoC-L columns). In the last two columns we report the difference in SLoC and functions between the input and output code (the delta columns). Note the woven and delta results for the HalfPrecisionOpenCL strategy are the sum of all generated code, totaling 31 versions.

An inspection of columns LARA SLoC-L and Delta SLoC-L reveals that, in most examples, there is a large overhead in terms of LARA SLoC-L over application SLoC-L. While this may seem a problem, we need to consider that a large part of the work being performed by these strategies is code analysis, which does not translate directly to SLoC-L in the final application. Furthermore, the Delta SLoC-L metric does not account for removed application code and for these cases a metric based on the similarity degree among code versions could be of more interest. Also, in real-world applications, the ratio of LARA SLoC-L to application SLoC-L would be definitely more favorable, thanks to aspect reuse.

To better understand the impact of analysis, we report in the first two columns of Table 2 the number of code points and of their attributes analysed, which can be compared with the last three column of the

Table 1: Static Metrics

Strategy	LARA SLoC-L	LARA Aspects	Input SLoC-L	Input Func	Woven SLoC-L	Woven Func	Delta SLoC-L	Delta Func
ChangePrecision	27	1	12	3	13	3	1	0
SimpleExamon	20	1	12	3	23	5	11	2
Multiversion	46	2	12	3	43	5	31	2
CreateFloatVersion	28	2	12	3	24	3	12	0
SimpleLibVC	12	1	12	3	39	4	27	1
HalfPrecisionOpenCL*	93	3	9	1	279	31	270	30
Total	226	10	69	16	421	51	352	35

Table 2: Dynamic Metrics

File	Selects	Attributes	Actions	Inserts	Native SLoC
ChangePrecision	4	109	2	1	0
SimpleExamon	4	131	18	7	0
Multiversion	8	477	27	16	9
HalfPrecisionOpenCL	125	2211	381	159	31
CreateFloatVersion	2	170	6	3	0
SimpleLibVC	7	93	13	8	36
Total	150	3191	447	194	76

same table, which instead report the corresponding effects, in terms of the number of modified points and lines of code inserted. To understand the impact of removed lines of code, we look at the *Inserts* and *Actions* columns, which show that circa one half of the actions do not insert code. The end line is that the analysis work exceeds the transformation work by an order of magnitude, and the insertions only understimate significantly the work performed.

Another benefit for user productivity when using LARA is how the techniques presented in the examples can scale into large-scale applications and scenarios. Most of the presented strategies are parameterized by function, i.e., they receive a function join point or name and act on the corresponding function. This could be performed manually, albeit crudely, using a search function of an IDE. Consider the case where we instead want to target a set of functions, whose names we may not know, based on their function signatures, or based on the characteristics of the variables declared inside their scope. This kind of search and filtering based on syntactic and semantic information available in the program is one of the key features of LARA and it cannot be easily attained with other tools. As the aspects presented here illustrate, LARA strategies can be made reusable and applied over large applications, greatly out scaling the effort needed to develop them.

6 Conclusions

To fully exploit the heterogeneous resources of future Exascale HPC systems, new software stacks are needed to provide power management, optimization, and autotuning to the parallel applications deployed on such systems. The ANTAREX project provides a holistic system-wide adaptive approach

for next generation HPC systems, centered around a domain specific language that allows a full decoupling of functional and extra-functional specifications for each application, providing integration with a wide range of support tools. We have shown how the ANTAREX tool flow allows developers to control the precision of a computation, to manage dynamic code specialization, monitoring, power capping, and dynamic autotuning. The impact and benefits of such technology are far reaching, beyond traditional HPC domains.

References

[1] J. Floch *et al.*, "Using architecture models for runtime adaptability," *IEEE Softw.*, vol. 23, no. 2, pp. 62–70, Mar. 2006.

[2] J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, and J.-M. Loingtier, "Aspect-oriented Programming," in *ECOOP'97 – Object-Oriented Programming*, ser. LNCS. Springer, 1997, vol. 1241, pp. 220–242.

[3] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented Programming: Introduction," *Commun ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[4] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: An Aspect-oriented Programming Language for Embedded Systems," in *Proc. 11th Annual Int'l Conf. on Aspect-oriented Software Development*. ACM, 2012, pp. 179–190.

[5] J. M. P. Cardoso, J. G. F. Coutinho, T. Carvalho, P. C. Diniz, Z. Petrov, W. Luk, and F. Gonćalves, "Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach," *Software: Practice and Experience*, Dec. 2014.

[6] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-oriented Extension to the C++ Programming Language," in *Proc. 40th Int'l Conf on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, 2002, pp. 53–60.

[7] J. M. Cardoso, T. Carvalho, J. G. Coutinho, R. Nobre, R. Nane, P. C. Diniz, Z. Petrov, W. Luk, and K. Bertels, "Controlling a complete hardware synthesis toolchain with LARA aspects," *Microprocessors and Microsystems*, vol. 37, no. 8, pp. 1073–1089, 2013.

[8] R. Nobre, L. G. Martins, and J. M. Cardoso, "Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration," in *Proc. of Int'l Workshop on Software and Compilers for Embedded Systems*. ACM, 2015, pp. 58–67.

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

[10] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu, "Evaluating iterative optimization across 1000 datasets," in *Proc 31st ACM SIGPLAN Conf on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2010, pp. 448–459.

[11] M. Tartara and S. Crespi Reghizzi, "Continuous learning of compiler heuristics," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 46:1–46:25, Jan. 2013.

[12] A. H. Ashouri, G. Mariani *et al.*, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 21:1–21:25, Jun. 2016.

[13] A. H. Ashouri, A. Bignoli *et al.*, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 29:1–29:28, Sep. 2017.

[14] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicore architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 Int'l Conf on*. IEEE, 2015, pp. 173–180.

[15] Y. Brun *et al.*, *Engineering Self-Adaptive Systems through Feedback Loops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_3

[16] D. Gadioli *et al.*, "Socrates - a seamless online compiler and system runtime autotuning framework for energy-aware applications," in *Proc. of Design Automation and Test in Europe (DATE18)*, 2018, pp. 1149–1152.

[17] F. Beneventi, A. Bartolini, C. Cavazzoni, and L. Benini, "Continuous learning of hpc infrastructure models using big data analytics and in-memory processing tools," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 1038–1043.

[18] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug 2010, pp. 189–194.

[19] D. Cesarini, A. Bartolini, and L. Benini, "Benefits in relaxing the power capping constraint," in *ANDARE '17*. ACM, 2017, pp. 3:1–3:6.

[20] A. Bartolini, R. Diversi, D. Cesarini, and F. Beneventi, "Self-aware thermal management for high performance computing processors," *IEEE Design & Test*, 2017.

[21] D. Cesarini, A. Bartolini, and L. Benini, "Prediction horizon vs. efficiency of optimal dynamic thermal control policies in hpc nodes," in *2017 IFIP/IEEE Int'l Conf on Very Large Scale Integration (VLSI-SoC)*, Oct 2017, pp. 1–6.

[22] M. Golasowski, J. Bispo, J. Martinovič, K. Slaninová, and J. M. Cardoso, "Expressing and applying c++ code transformations for the hdf5 api through a dsl," in *IFIP Int'l Conf on Computer Information Systems and Industrial Management*. Springer, 2017, pp. 303–314.

[23] M. Golasowski, R. Tomis, J. Martinovič, K. Slaninová, and L. Rapant, "Performance evaluation of probabilistic time-dependent travel time computation," in *IFIP Int'l Conf on Computer Information Systems and Industrial Management*. Springer, 2016, pp. 377–388.

[24] M. K. L. Zadnik Stirn, Ed., *Server-side navigation service benchmarking tool*, Slovenian Society Informatika. Litostrojska cesta 54, Ljubljana, Slovenia: Slovenian Society Informatika, Sept 2017.

```
For two versions of FOO and 2 set of compiler options, called in a loop such that

  for (i=0; i< N; i++) {
  FOO();
}

it is transformed into (we assume that N is a multiple of 4)

// Running the first version, first options
   LOADING_BIB(libV0);
   t0_begin = getCurrentTime();
   for (i=0; i< N/4; i++) {
     FOO();
   }
   t0_end = getCurrentTime();
   t0 = t0_end - t0_begin;

// Running the first version, second options
   LOADING_BIB(libV1); //
   t1_begin = getCurrentTime();
   for (i=(N/4)+1; i< 2*(N/4); i++) {
     FOO();
   }
   t1_end = getCurrentTime();
   t1 = t1_end - t1_begin;

// Running the second version, first options
   LOADING_BIB(libV2);
   t2_begin = getCurrentTime();
   for (i= 2*(N/4)+1; i< 3*(N/4); i++) {
       FOO();
   }
   t2_end = getCurrentTime();
   t2 = t2_end - t2_begin;

// Running the second version, second options
   LOADING_BIB(libV3);
   t3_begin = getCurrentTime();
   for (i=3*(N/4)+1; i<N; i++) {
      FOO();
   }
   t3_end = getCurrentTime();
   t3 = t3_end - t3_begin;

// Take the decision: best solution.
   ...
```

Figure 12: Split compiling principles.

```
aspectdef mARGOt_Aspect
/* Input: TargetFunctionCall*/
input targetCallName end
/* mARGOt configuration */
  var config = new MargotConfig();
  var targetBlock = config.newBlock($targetCallName);
  targetBlock.addKnob('Knob1', 'knob1', 'int');
  targetBlock.addMetric('error', 'float');
  targetBlock.addMetric('throughput', 'float');
  targetBlock.addMetricGoal('my_error_goal',
    MargotCFun.LE, 0.03, 'error');

  /* optimization problem */
  var problem = targetBlock.newState('defaultState');
  problem.maximizeMetric('throughput');
  problem.subjectTo('my_error_goal');

  /* generate the information needed
     for enhancing the application code */
  codegen = MargotCodeGen.fromConfig(config, $targetCallName);

/* Target function call identification */
select stmt.call{targetName} end

/* Add mARGOt calls*/
codegen.init($call);
codegen.update($call);
end
```

Figure 13: Example of a LARA aspect for autotuner configuration and code enhancement.

```
import antarex.examon.Examon;
import lara.code.Logger;

aspectdef SimpleExamon

  input
    name, ip, topic, $manageFunction, $targetStmt
  end

  var broker = new ExamonBroker(ip);

  var exa = new ExamonCollector(name, topic);

  // manage the collector on the target function
  select $manageFunction.body end
  apply
    exa.init(broker, $body);
    exa.start($body);

    exa.end($body);
    exa.clean($body);
  end

  // get the value and use it in the target stmt
  exa.get($targetStmt);

  // get the last stmt of the scope of the target stmt
  var $lastStmt = $targetStmt.ancestor("scope").lastStmt;
  // Create printf for time and data
  var logger = new Logger();
  logger.ln().text("Time=").double(getTimeExpr(exa))
    .text("[s], data=").double(exa.getMean()).ln();
  // Add printf after last stmt
  logger.log($lastStmt);
end
```

Figure 14: Example of a LARA aspect integrate an ExaMon collector into an application.

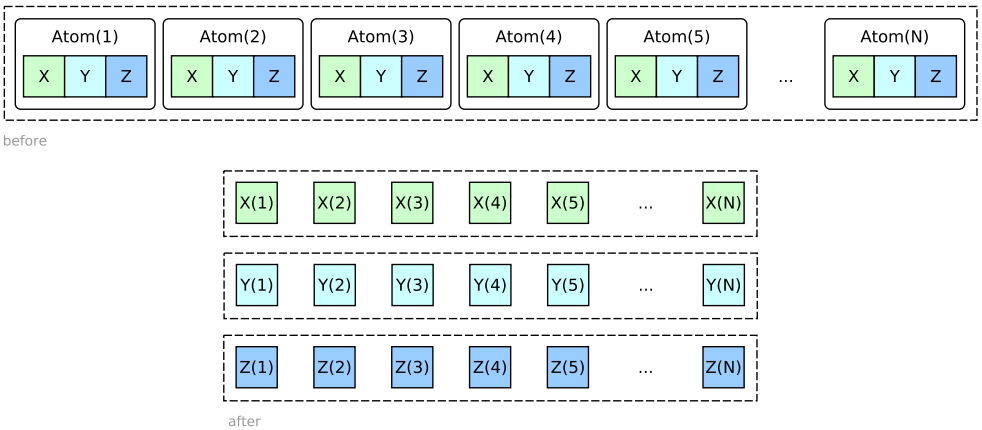

Figure 15: Conversion of an array of atoms into an three arrays of coordinates.

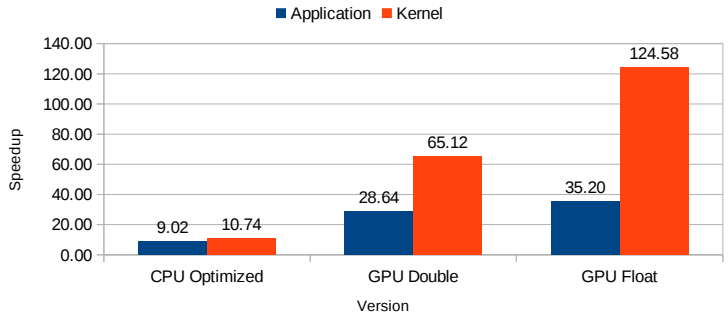

Figure 16: Speedups of the developed versions in relation to the original UC1 application.

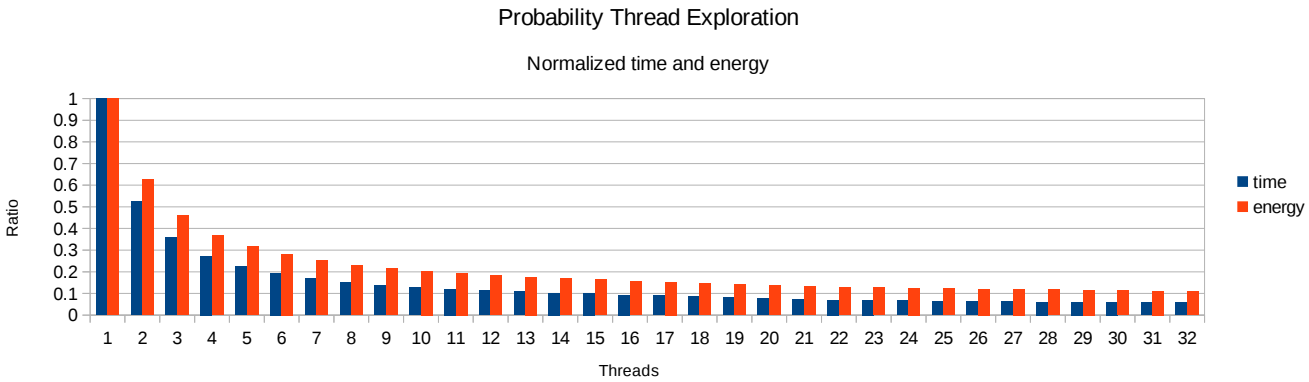

Figure 17: Execution times and energy consumed for every tested number of threads normalized against using 1 thread.

libVersioningCompiler: an easy-to-use library for dynamic generation and invocation of multiple code versions

S. Cherubin, G. Agosta

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano
Via G. Ponzio 34/5, I-20133 Milano, Italy

Abstract

We present libVersioningCompiler, a C++ library designed to support the dynamic generation of multiple versions of the same compute kernel in a HPC scenario. It can be used to provide continuous optimization, code specialization based on the input data or on workload changes, or otherwise to dynamically adjust the application, without the burden of a full dynamic compiler. The library supports multiple underlying compilers but specifically targets the llvm framework.

We also provide examples of use, showing the overhead of the library, and providing guidelines for its efficient use.

Keywords: Dynamic Compilation, Versioning Compiler, Continuous Optimization

1. Motivation and significance

Designing and implementing High Performance Computing (HPC) applications is a difficult and complex task that requires mastery of several specialized languages and performance-tuning tools; however, this prerequisite is incompatible with the current trend that opens HPC infrastructures to a wider range of users [1, 2]. The current model that sees the HPC center staff directly supporting the development of applications will become unsustainable in the long term. Thus, the availability of effective APIs and programming languages is crucial to provide migration paths towards novel

Email address: stefano.cherubin@polimi.it (S. Cherubin)

heterogeneous HPC platforms as well as to guarantee the developers' ability to work effectively on these platforms.

Profile-guided code transformations at compile-time usually provide a good optimization level in a general-purpose scenario. On the contrary, in HPC scenarios where large data sets are employed, a proper profiling may be unfeasible. In these cases, which are becoming more and more common [3], dynamic approaches can prove more effective. The practice of improving the application code at runtime through dynamic recompilation is known as *continuous program optimization* [4, 5, 6]. Although it has been studied for more than a decade, very few people adopt it in practice since it is difficult to perform manually, and, when performed automatically, it can compromise software maintainability. At the same time, autotuning is used both to tune software parameters and to search the space of compiler optimizations for optimal solutions [7]. Autotuning frameworks can select one of a set of different versions of the same computational kernel to best fit the HPC system runtime conditions, such as system resource partitioning, as long as such versions are generated at compile time. Few of these frameworks are actually able to perform continuous optimization, and those that support it do so only through specific versions of a dynamic compiler [8, 9] or through cloud-based platforms [10].

libVersioningCompiler (abbreviated libVC) can be used to perform continuous program optimization using simple C++ APIs. libVC allows different versions of the executable code of a computational kernel to be transparently generated on the fly. Continuous program optimization with libVC can be performed by dynamically enabling or disabling code transformations, and changing compile-time parameters according to the decisions of other software tools such as a generic application autotuner.

The rest of the paper is organized as follows. In section 2 we describe the software architecture, the internal APIs and their functionalities. In section 3 we introduce an example of intended use and discuss benefits and overhead deriving from the implementation of continuous program optimization through libVC in a generic scenario. In section 4 we highlight the impact of libVC in both industry and research field. Finally, we draw some conclusions in section 5.

2. Software description

The goal of libVC is to allow a C/C++ compute kernel to be dynamically compiled multiple times while the program is running, so that different specialized versions of the code can be generated and invoked. This capability is especially useful when the optimal parametrization of the compiler

depends on the program workload. In these cases, the ability to switch at runtime between different versions of the same code can provide significant benefits, as shown in [11, 12].

Indeed, in general-purpose code it is preferable to profile the application to statically generate the most efficient versions ahead of time. However, in HPC code the execution times are usually so long that a profiling run may not be an attractive choice. On the contrary, LIBVC enables the exploration and tuning of the parameter space of the compiler at runtime, while the program is performing useful work.

LIBVC considers as valid compute kernels any C-like procedure or function that can be compiled to object code. There is just one constraint that should be enforced on the compute kernel: it must respect C linkage rules. This means that no name mangling should be applied to the compute kernel itself. Within our model, the `Compiler` is the tool used to compile the compute kernel, and the `Version` is the configuration passed to the compilation task. We assume to work with deterministic `Compiler`s. In this scenario, a `Version` produces at most one executable code. No executable code is generated when the specified configuration is invalid.

2.1. Software Architecture

The LIBVC source code is available under the LGPLv3 licence. It is compliant with the C++11 standard and it comes with configuration files to ease the setup by using the `CMake` build system. The minimum required `CMake` version is 3.0.2. The build system automatically checks the presence of the optional dependencies `LLVM` and `libClang`, whose version must be greater than `4.0.0`. Whenever these dependencies are not satisfied, some features are automatically disabled during the library installation. Please see table 2 on page 14 for a detailed and exhaustive list of dependencies.

Description of the software model. Figure 1 shows a simplified UML class diagram of this software. It is possible to identify three main classes in the source code. The simplest class, which is called `Option`, represents each of the flag and parameters that are passed to LibVC in order to compile a version of a computing kernel. The `Compiler` abstract class defines the interface that allows the host application to interact with `Compiler` implementations. LIBVC provides up to three possible implementations for the `Compiler` abstract class: `SystemCompiler`, which relies on system calls to external compilers that are already installed in the host system; `SystemCompilerOptimizer`, which is an extension of a `SystemCompiler` that also supports external optimization tools (such as the LLVM optimizer `opt`); and `ClangLibCompiler`,

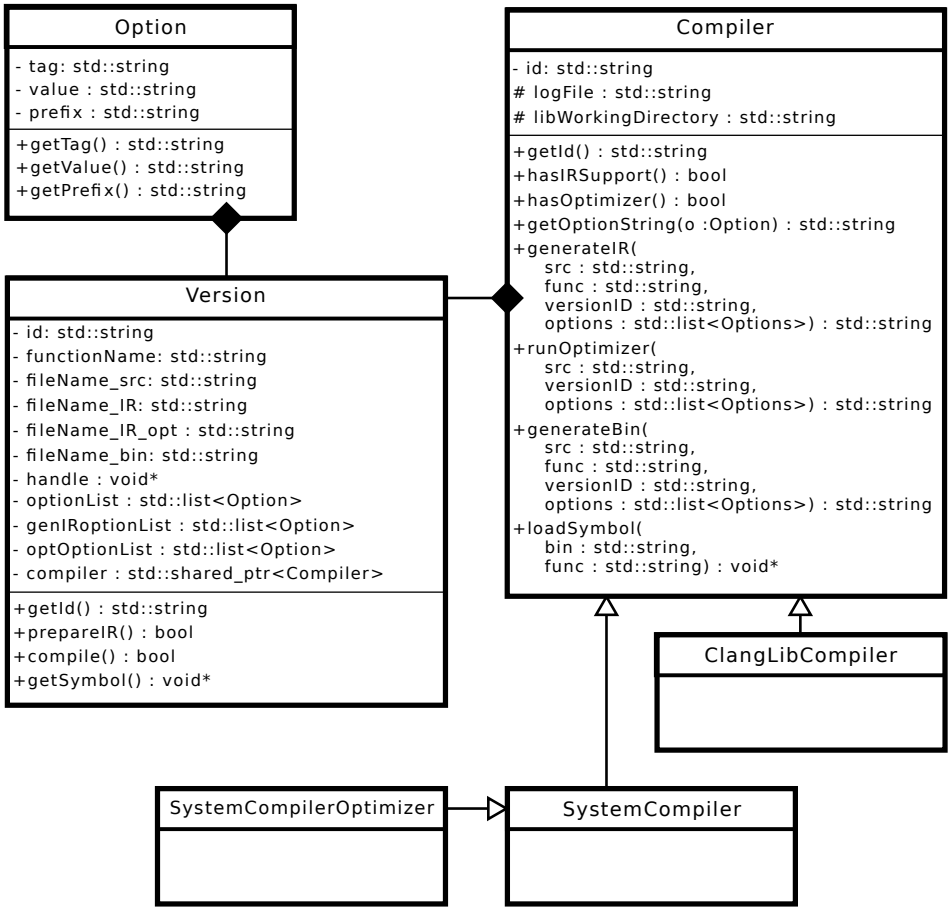

Figure 1: Simplified UML class diagram of libVC

which exploits the compiler-as-a-library paradigm through the `Clang` APIs[1]. Please note that `ClangLibCompiler` is installed only if `LLVM` and `libClang` dependencies are satisfied. The last important class is the `Version` class, which represents a compute kernel defined in a specific source file, with a given compiler configuration. A `Version` object is compiled with the chosen `Compiler` using an ordered list of `Options`. It contains a unique identifier, references to `Compiler` and `Options` used to compile it, and references to the files that are generated by the `Compiler` while compiling the `Version`. The configuration of a `Version` object is immutable throughout the lifetime

[1] `http://clang.llvm.org/docs/Tooling.html`

of that object. The `Version` class also provides APIs to control the stages of the compilation process: it is possible to create a `Version` object and postpone the execution of the selected `Compiler` to a later stage.

2.2. Software Functionalities

libVC provides an easy-to-use interface that can be employed to perform the dynamic compilation of a kernel, and to load compiled `Version`s as C-like function pointers. libVC itself does not provide any automatic selection of which `Version` should be executed. The decision of which `Version` is the most suitable for a given task is left to policies defined by the programmer or other autotuning frameworks such as mARGOt [13] or cTuning [14].

libVC comes in two different flavours: with detailed low-level APIs and with simple high-level APIs. The latter is optimized for the most common use cases, they exploit the default system compiler and do not support any external optimization tool, whereas low-level APIs allow a more fine grained setup and support split-compilation techniques [15]; hence, the resulting source code is slightly more verbose.

The typical usage of libVC involves different stages. The first task must be the declaration and initialization of the `Version`-independent tools, such as `Compiler`s and `Version` builders, which are helper objects designed to properly setup a `Version` configuration. Low-level APIs allow the programmer to customize one or more `Compiler` implementations. High-level APIs expose a special function to transparently perform this task; it is required to be invoked just once in the whole process lifetime. After that, it is possible to proceed to the `Version` configuration. The programmer can, by using low-level APIs, dynamically forge and arrange `Option`s, set the chosen `Compiler`s, manipulate file and kernel names to identify the code to be compiled. The `Version` builder is the component which allows this low-level setup. Once the `Version` builder has its fields filled up, it can be finalized to generate a `Version` object. High-level APIs receive all these parameters and produce a `Version` object in a single function call. High-level APIs limit the configuration to one `Version` at a time while low-level APIs allows parallel configuration of multiple `Version`s. Once a `Version` object is finalized, it has to be compiled. The compilation task is activated by the programmer through a dedicated API. It may trigger more than one sub-task when it involves split-compilation techniques. In the absence of compilation errors, and regardless of which APIs are being used, at the end of this stage libVC generates a binary shared object. From this same shared object libVC loads a function pointer symbol, which points to the kernel.

The target kernel may include other files or refer to external symbols. libVC will act just as a compiler invocation and will try to resolve external

symbols according to the given compiler and linker options.

LIBVC defers the resolution of the compilation parameters to run-time. The only piece of information that is needed at design-time is the prototype of the kernel, which has to be used for a proper function pointer cast.

LIBVC also provides hooks to enable tracking and versioning of the compiled versions.

3. Illustrative Examples

LIBVC can be exploited to apply a wide range of optimization through the dynamic compilation. The official repository[2] provides some examples of usage in the test files. In this section we show and discuss a generic use case of continuous program optimization performed through LIBVC. Listing 1 illustrates the dynamic adaptation of a counting sort algorithm to the data workload. In particular, the counting sort implementation is specialized through recompilation using LIBVC every time the `min` and `max` value of range of the data to be sorted change. When the `min` and `max` values of the range of the data are known at compile-time it is possible to perform array allocation and loop optimizations more efficiently.

Listing 1: Benchmark of a statically linked kernel performing counting sort against a dynamically compiled version of the same kernel using LIBVC high-level APIs

```
1  // libVersioningCompiler High-Level API header file
2  #include "versioningCompiler/Utils.hpp"
3
4  // define kernel signature
5  typedef void (*kernel_t)(std::vector<int32_t> &array);
6
7  vc::version_ptr_t getDynamicVersion(int32_t min, int32_t max) {
8    // version configuration using libVC - start
9    const std::string kernel_dir = PATH_TO_KERNEL;
10   const std::string kernel_file = kernel_dir + "kernel.cpp";
11   const std::string functionName = "vc_sort";
12   const vc::opt_list_t opt_list = {
13     vc::make_option("-O3"),
14     vc::make_option("-std=c++11"),
15     vc::make_option("-I"+kernel_dir),
16     vc::make_option("-D_MIN_VALUE_RANGE="+std::to_string(min)),
17     vc::make_option("-D_MAX_VALUE_RANGE="+std::to_string(max)),
18   };
19   vc::version_ptr_t version = vc::createVersion(kernel_file,
       functionName, opt_list);
20   // version configuration using libVC - end
```

[2] https://github.com/skeru/libVersioningCompiler

```
21
22    // version compilation - start
23    kernel_t f = (kernel_t) vc::compileAndGetSymbol(version);
24    if (f) {
25      return version;
26    }
27    // version compilation - end
28    return nullptr;
29  }
30
31  int main(int argc, char const *argv[]) {
32    const std::vector<std::pair<int, int>> data_range = {
33      std::make_pair<int,int>(0,256),
34      std::make_pair<int,int>(0,512),
35      std::make_pair<int,int>(0,1024),
36    };
37    const size_t data_size = 1000000000;
38
39    // initialize libVersioningCompiler
40    vc::vc_utils_init();
41
42    for (const auto range : data_range) {
43      TimeMonitor time_monitor_ref;
44      TimeMonitor time_monitor_dyn;
45      TimeMonitor time_monitor_ovh;
46
47      // running reference version - statically linked
48      for (size_t i = 0; i < iterations; i++) {
49        // produce workload to process
50        auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(
              range.first, range.second);
51        const auto meta = wl.getMetadata();
52        time_monitor_ref.start();
53        sort(wl.data, meta.minVal, meta.maxVal); // call reference
54        time_monitor_ref.stop();
55      }
56
57      // measuring overhead of preparing a new version - start
58      time_monitor_ovh.start();
59      auto v = getDynamicVersion(range.first, range.second);
60      kernel_t my_sort = (kernel_t) v->getSymbol();
61      time_monitor_ovh.stop();
62      // measuring overhead of preparing a new version - end
63
64      // running dynamic version - dynamically compiled
65      for (size_t i = 0; i < iterations; i++) {
66        // produce workload to process
67        auto wl = WorkloadProducer<int32_t>::get_WL_with_bounds(
              range.first, range.second);
```

```
68          time_monitor_dyn.start();
69          my_sort(wl.data); // just a call to a function pointer
70          time_monitor_dyn.stop();
71        }
72
73        // consider average time-to-solution
74        std::cout << range.second << " " << time_monitor_ref.getAvg
              () << " " << time_monitor_dyn.getAvg() << " " <<
              time_monitor_ovh.getAvg() << std::endl;
75      }
76      return 0;
77  }
```

Listing 1 reveals the several stages of the compilation flow of libVC. In the `main` function, an initialization is needed before using libVC. This is done in line 40 using a simple API invocation. From line 8 to line 20 we see how to configure a new `Version` for dynamic compilation. The following lines (22 - 27) perform the actual dynamic compilation. It is possible to notice in line 69 the call to the dynamically compiled kernel, which is very similar to the call to a statically linked kernel (line 53).

As proof of concept, we tested the benefits of continuous program optimization implemented with libVC by comparing the time-to-solution of the statically linked kernel against a dynamically compiled version of the same kernel, as shown in listing 1. We compiled both the statically linked and the dynamically compiled kernels using the same compiler and the same optimization level. A full project using code from listing 1 is available on github[3]. We run this example to sort an array of 1 billion 32-bits integers. The platform used to execute the experiment is a supercomputer NUMA node that features two Intel Xeon E5-2630 V3 CPUs (@2.4 GHz) with 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration.

Table 1 shows that dynamically compiled kernels always performs better with respect to the reference statically linked implementation. We define as range size the difference between `max` and `min` values of the range of the data to be sorted. We observe an important speedup when the range size is smaller than 8192 possible values. In those cases the main part of the speedup comes from a more efficient memory allocation of the array in the dynamically compiled kernels. We also notice that the overhead of dynamically compiling a new `Version` is not related with the range size. This overhead can be absorbed within 3 iterations when the range size is small, and within less than one thousand iterations in the worst case.

[3] `https://github.com/skeru/countingsort_libVC`

Range size [elements]	TTS reference [ms]	TTS libVC [ms]	speedup [%]	overhead [ms]	payback [iterations]
256	2831.33	2368.12	19.56	1355.99	3
512	2822.84	2352.27	20.00	1345.25	3
1024	2820.67	2347.28	20.17	1356.86	3
2048	2831.92	2351.99	20.41	1361.37	3
4096	2914.13	2440.47	19.41	1353.05	3
8192	3967.59	3966.21	0.03	1354.12	982
16384	5168.64	5163.51	0.10	1370.82	268
32768	6459.75	6430.77	0.45	1358.26	47

Table 1: Experimental results of Time-To-Solution (TTS) averaged over 100 executions on a Ubuntu x86_64 system. Kernels were compiled using `gcc` 5.4.0 with optimization level `-O3`.

It is also possible to use libVC to dynamically compile and run several functions or the same function with different options. A more complex example of usage of libVC which exploits these features can be found on github[4] where we dynamically compile and run the full PolyBench/C [16] benchmark suite within the same C++ program.

4. Impact

libVC is a software tool that supports the generation and execution of multiple versions of C++ kernels. This means that libVC allows a wider range of users to adopt continuous optimization practices by generating workload-dependent specializations of one or more kernels. Accordingly, libVC enables the development of autotuning techniques, as well as the comparison of different autotuning algorithms within a neutral platform with any desired compiler. By providing the option to select multiple compilers, libVC can be easily adopted by industrial users, such as supercomputing centers, as they are often constrained to vendor-specific compilers.

libVC is used within the European project ANTAREX [17, 18], which aims at expressing the capability of applications to self-adapt to runtime conditions (we call this practice *autotuning*) through a Domain Specific Language (DSL) and at providing runtime management and autotuning support for applications that target green and heterogeneous HPC systems up to Exascale. The application functionality is expressed through C/C++ code

[4] `https://github.com/skeru/polybench_libVC`

(possibly including legacy code), whereas the non-functional aspects of the application, including parallelization, mapping, and adaptivity strategies are expressed through the DSL developed in the project. The application autotuning is delayed to the runtime phase, where the *software knobs* (application parameters, code transformations and code variants) are configured according to the runtime information that is retrieved from the execution environment. LIBVC serves to dynamically provide code transformations and code variants in the ANTAREX tool flow. The ANTAREX consortium includes two major European supercomputing centers, as well as industrial users in the automotive and bioinformatics application domains.

Case Study: Geometrical Docking Miniapp. To assess the impact of the proposed tools on a real-world application we employ a miniapp developed within the ANTAREX project [17] to emulate the workload of the geometric approach to molecular docking. This class of application is useful in the in-silico drug-discovery process, which is an emerging application of HPC, and consists in finding the best fitting ligand molecule with a pocket in the target molecule [19]. This process is performed by approximating the chemical interactions with the proximity between atoms.

We processed a database of 113161 ligand molecule-pocket pairs on the same test platform we describe in section 3. The evaluation of every ligand molecule-pocket pair is independent with respect to the other pairs. Therefore, we implemented an MPI-based version of the same miniapp. The input dataset is partitioned among the slave processes.

The initial code base was not developed by the authors.We integrated the code which is executed by each slave process with LIBVC, as for the serial version. It took one hour of work to integrate the miniapp source code with the LIBVC. The integration required to add or modify a total of 60 lines of code over an original code size of 1300 lines of code, which is less than 5% of the code size.

The baseline miniapp took 4354.95 seconds before the integration. After the integration the miniapp took 1783.93 seconds – including the overhead for dynamic compilation – for a speedup of 2.44× with respect to the baseline. The speedup is achieved by exploiting code specialization on geometrical functions.

Although the overhead of performing dynamic compilation on every parallel process slows down the running time, the speedup we obtained in the serial version of the miniapp is confirmed also in the parallel case. We run the MPI-based miniapp using 4, 8, 16, and 32 parallel processes. We obtained a speedup of 2.39×, 2.24×, 1.99×, and 1.63× respectively.

Case Study: OpenModelica Compiler. To assess the impact of the proposed tools on a legacy code we employ the C code which is automatically generated by a state-of-the-art compiler for Modelica. Modelica is a widely-used object-oriented language for modeling and simulation of complex systems. OpenModelica [20] is an open source compiler for the Modelica language. It translates Modelica code into C code, which is later compiled with `clang` and linked against an external equation solver library.

As test case, we simulated a transmission line model [21] of 1000 elements. We modified the C and Makefile code automatically generated by the OpenModelica compiler to integrate the simulation C source code with libVC and properly compile it. It took two hours of work to integrate the automatically generated code with the libVC. The integration required to add or modify a total of 65 lines of C code and 5 lines of Makefile code over an original code size of 633390 lines of code, which is less than 0.015% of the code size.

The baseline code took 374.25 seconds before the integration. After the integration the simulation took 295.00 seconds – including the overhead for dynamic compilation – for a speedup of 1.27× with respect to the baseline. The speedup is achieved by recompiling the C code which implements the model description by using a deeper optimization level (`-O3`) with respect to the default one (`-O0`). In this case, the compilation time that it is spent on optimizations is widely paid back by a faster execution time.

5. Conclusions

We have presented libVC, a lightweight library to support continuous optimization in HPC environments. The tool is employed within the context of the ANTAREX project to optimize the execution of computationally intensive kernels that are repeatedly called within large scale applications with long execution times. While the library is designed to be integrated with other tools in the ANTAREX workflow, it can also be used as a standalone tool with minimal effort by application developers.

Acknowledgements

This work is partially supported by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 671623, FET-HPC ANTAREX.

The authors wish to thank Emanuele Baldino, and Francesco Casella from Politecnico di Milano for providing the Modelica source code of the Transmission Line example employed in section 4.

[1] W. Ziegler, R. D'Ippolito, M. D'Auria, J. Berends, M. Nelissen, R. Diaz, Implementing a "one-stop-shop" providing smes with integrated hpc simulation resources using fortissimo resources, in: eChallenges e-2014 Conference Proceedings, 2014, pp. 1–11.

[2] B. Koller, N. Struckmann, J. Buchholz, M. Gienger, Towards an environment to deliver high performance computing to small and medium enterprises, in: Sustained Simulation Performance 2015, Springer, 2015, pp. 41–50.

[3] D. A. Reed, J. Dongarra, Exascale computing and big data, Communications of the ACM 58 (7) (2015) 56–68. doi:10.1145/2699414.

[4] T. Kistler, M. Franz, Continuous program optimization: A case study, ACM Trans. Program. Lang. Syst. 25 (4) (2003) 500–548. doi:10.1145/778559.778562.

[5] D. Nuzman, R. Eres, S. Dyshel, M. Zalmanovici, J. Castanos, Jit technology with c/c++: Feedback-directed dynamic recompilation for statically compiled languages, ACM Trans. Archit. Code Optim. 10 (4) (2013) 59:1–59:25. doi:10.1145/2541228.2555315.

[6] B. Fahs, T. Rafacz, S. J. Patel, S. S. Lumetta, Continuous optimization, in: Proceedings of the 32Nd Annual International Symposium on Computer Architecture, ISCA '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 86–97. doi:10.1109/ISCA.2005.19.

[7] S. Benkner, F. Franchetti, H. M. Gerndt, J. K. Hollingsworth, Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401), Dagstuhl Reports 3 (9) (2014) 214–244. doi:10.4230/DagRep.3.9.214.

[8] H. Chen, J. Lu, W.-C. Hsu, P.-C. Yew, Continuous adaptive object-code re-optimization framework, in: P.-C. Yew, J. Xue (Eds.), Advances in Computer Systems Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 241–255. doi:10.1007/978-3-540-30102-8_20.

[9] P. Basu, S. Williams, B. V. Straalen, L. Oliker, P. Colella, M. Hall, Compiler-based code generation and autotuning for geometric multigrid on gpu-accelerated supercomputers, Parallel Computing 64 (Supplement C) (2017) 50 – 64, high-End Computing for Next-Generation Scientific Discovery. doi:10.1016/j.parco.2017.04.002.

[10] J. Cohen, T. Rayna, J. Darlington, Understanding resource selection requirements for computationally intensive tasks on heterogeneous computing infrastructure, in: J. Á. Bañares, K. Tserpes, J. Altmann (Eds.), Economics of Grids, Clouds, Systems, and Services, Springer International Publishing, Cham, 2017, pp. 250–262. doi:10.1007/978-3-319-61920-0_18.

[11] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, C. Wu, Evaluating iterative optimization across 1000 datasets, in: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 448–459. doi:10.1145/1806596.1806647.

[12] M. Tartara, S. Crespi Reghizzi, Continuous learning of compiler heuristics, ACM Trans. Archit. Code Optim. 9 (4) (2013) 46:1–46:25. doi:10.1145/2400682.2400705.

[13] D. Gadioli, G. Palermo, C. Silvano, Application autotuning to support runtime adaptivity in multicore architectures, in: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on, IEEE, 2015, pp. 173–180.

[14] G. Fursin, A. Lokhmotov, E. Plowman, Collective Knowledge: towards R&D sustainability, in: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16), 2016, pp. 864–869.

[15] A. Cohen, E. Rohou, Processor virtualization and split compilation for heterogeneous multicore embedded systems, in: Design Automation Conference, 2010, pp. 102–107. doi:10.1145/1837274.1837303.

[16] T. Yuki, Understanding PolyBench/C 3.2 kernels, in: International workshop on Polyhedral Compilation Techniques (IMPACT), 2014.

[17] C. Silvano, G. Agosta, S. Cherubin, D. Gadioli, G. Palermo, A. Bartolini, L. Benini, J. Martinovič, M. Palkovič, K. Slaninová, et al., The antarex approach to autotuning and adaptivity for energy efficient hpc systems, in: Proceedings of the ACM International Conference on Computing Frontiers, CF '16, ACM, New York, NY, USA, 2016, pp. 288–293. doi:10.1145/2903150.2903470.

[18] C. Silvano, G. Agosta, A. Bartolini, A. R. Beccari, L. Benini, J. Bispo, R. Cmar, J. M. Cardoso, C. Cavazzoni, J. Martinovič, et al., Autotuning and adaptivity approach for energy efficient exascale hpc systems: the

antarex approach, in: Proceedings of the 2016 Conference on Design, Automation & Test in Europe, DATE '16, 2016, pp. 708–713.

[19] A. R. Beccari, C. Cavazzoni, C. Beato, G. Costantino, Ligen: a high performance workflow for chemistry driven de novo design (2013).

[20] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, A. Sandholm, Openmodelica - a free open-source environment for system modeling, simulation, and teaching, in: 2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006, pp. 1588–1595. doi:10.1109/CACSD-CCA-ISIC.2006.4776878.

[21] F. Casella, Simulation of large-scale models in modelica: State of the art and future perspectives, in: LINKÖPING ELECTRONIC CONFERENCE PROCEEDINGS, 2015, pp. 459–468.

Required Metadata

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	v1.2
C2	Permanent link to code/repository used for this code version	`https://github.com/skeru/libVersioningCompiler`
C3	Legal Code License	LGPL v3
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	C++, cmake, LLVM, Clang.
C6	Compilation requirements, operating environments & dependencies	Suggested system: Ubuntu 16.04 x86_64 or version greater. Required dependencies: dl, uuid-dev. Optional dependencies: llvm-4.0-dev, libclang-4.0-dev.
C7	If available Link to developer documentation/manual	See README.md in the repository
C8	Support email for questions	stefano.cherubin@polimi.it

Table 2: Code metadata (mandatory)