Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

The ANTAREX domain specific language for high performance computing

(Article begins on next page)

27 April 2024

The ANTAREX Domain Specific Language for High Performance Computing

Cristina Silvano[a], Giovanni Agosta[a], Andrea Bartolini[c], Andrea R. Beccari[d], Luca Benini[b], Loïc Besnard[j], João Bispo[e], Radim Cmar[i], João M. P. Cardoso[e], Carlo Cavazzoni[f], Daniele Cesarini[c], Stefano Cherubin[a], Federico Ficarelli[f], Davide Gadioli[a], Martin Golasowski[g], Antonio Libri[b], Jan Martinovič[g], Gianluca Palermo[a], Pedro Pinto[e], Erven Rohou[h], Kateřina Slaninová[g], Emanuele Vitali[a]

[a]DEIB – Politecnico di Milano, Piazza Leonardo da Vinci 32, Milano, Italy
[b]IIS – Eidgenössische Technische Hochschule Zürich
[c]Alma Mater Studiorum - University of Bologna
[d]Dompé Farmaceutici SpA
[e]FEUP – Universidade do Porto
[f]CINECA
[g]IT4Innovations, VSB – Technical University of Ostrava
[h]INRIA Rennes
[i]Sygic
[j]IRISA/CNRS

Abstract

The ANTAREX project relies on a Domain Specific Language (DSL) based on Aspect Oriented Programming (AOP) concepts to allow applications to enforce extra functional properties such as energy-efficiency and performance and to optimize Quality of Service (QoS) in an adaptive way. The DSL approach allows the definition of energy-efficiency, performance, and adaptivity strategies as well as their enforcement at runtime through application autotuning and resource and power management. In this paper, we present an overview of the key outcome of the project, the ANTAREX DSL, and some of its capabilities through a number of examples, including how the DSL is applied in the context of the project use cases.

Keywords: High Performance Computing, Autotuning, Adaptivity, DSL, Compilers, Energy Efficiency

1. Introduction

High Performance Computing (HPC) is a strategic asset vigorously pursued by both state actors, supra-national entities, and major industrial players in fields such as finance or oil & gas [1]. A veritable arms race has been waged for decades to build the fastest HPC machine, first in terms of pure floating pooint operations per second (FLOPS), then, with the growing difficulty of supplying power to large HPC centers, in terms of Flops/watt. Currently, the goal is to reach Exascale level (10^{18} FLOPS) within the $2023 - 24$ timeframe – with a $\times 1000$ improvement over Petascale, reached

in 2009. The current Top500[1] and Green500[2] lists are dominated respectively by IBM's Summit machine, installed at the Oak Ridge National Laboratory (USA), with over 200 PetaFlops, and Japan's Shoubu system B, with a power efficiency of 17.6 GFlops/watt[3]. To reach these levels of performance and power efficiency, heterogeneous computing architectures are critical, in particular through GPGPUs. Traditional NVIDIA GPGPUs are featured heavily in both lists, though Shoubu's ZettaScaler architecture uses PEZY's PEZY-SC2 instead. The dominance of heterogeneous systems in the Green500 list is expected to continue for the next coming years to reach the target of 20MW Exascale supercomputers set by the DARPA.

When considering the design, development, and deplyoment of HPC applications, each increase in HPC machine size and heterogeneity imposes an increased burden on the developer. Nowadays, few developers have both the domain knowledge and HPC expertise to produce a fully optimised application, and a naïvely parallelised application may easily incur in parallelisation bottlenecks and scalability issues. To address this issue, the current development model assumes that the domain experts do not attempt to optimise their applications on their own, but rely on help from HPC experts, typically part of the HPC center staff, to turn their baseline application into an HPC-enabled one. This task requires to handle parallelisation, offloading to heterogeneous resources, as well as performance analysis and tuning. For the latter task, it is worth noting that for large scale applications it is oftern difficult to make predictions on performance based on small-scale runs, so runtime autotuning is desirable.

While many approaches to programming languages and models for HPC attempt to simplify the development process by reducing or removing the need for the HPC expert, the recently completed ANTAREX project [2, 3] fully embraces this split development process, and aims at supporting it. To this end, we introduce the ANTAREX DSL, a Domain Specific Language (DSL) to express the application self-adaptivity and to runtime manage and autotune applications for green heterogeneous HPC systems up to the Exascale level. The ANTAREX DSL allows the introduction of a separation of concerns, where self-adaptivity and energy efficient strategies, managed by the HPC expert, are specified separately from the application functionalities, developed instead by the domain expert. The ANTAREX DSL is based on Aspect Oriented Programming (AOP) principles, embodied in the LARA language [4, 5], which effectively support the separation of concern without burdening the domain expert with the need to learn a new language (as they simply develop in C/C++). Since HPC experts already use a variety of scripts and tools to support their work, the ANTAREX DSL also provides a way to unify them under a single interface.

To demonstrate the effectiveness of the ANTAREX DSL and its associated tools, the project employs two use cases taken from highly relevant HPC application scenarios:

1. a biopharmaceutical application for drug discovery developed by one of the leading European companies in the field, Dompé, and deployed on the 1.21 PetaFlops

---

[1] www.top500.org, November 2018

[2] www.green500.org, November 2018

[3] Summit is also ranked 3rd in the Green500, whereas Shoubu is a comparatively smaller machine

heterogeneous NeXtScale Intel-based IBM system at CINECA;

2. a self-adaptive navigation system for smart cities developed by the top European navigation software company, Sygic, and deployed on the server-side on the 1.46 PetaFlops heterogeneous Intel® Xeon Phi™ based system provided by IT4Innovations National Supercomputing Center.

These scenario also reproduce the aforementioned development model, as each HPC user company is supported by the staff of the HPC center where the application is deployed.

The remaining partners of the ANTAREX Consortium comprise a wealth of expertise in DSL development, code optimisation, energy efficiency, and autotuning. Four top-ranked academic and research partners (Politecnico di Milano, ETHZ Zurich, University of Porto and INRIA) are complemented by the Italian Tier-0 Supercomputing Center (CINECA), the Tier-1 Czech National Supercomputing Center (IT4Innovations) and two industrial application providers, one of the leading biopharmaceutical companies in Europe (Dompé) and the top European navigation software company (Sygic). Politecnico di Milano, the largest Technical University in Italy, played the role of Project Coordinator.

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.

Organization of the paper. The rest of this paper is organized as follows. In Section 2 we review the technology portfolio provided by the ANTAREX tool flow. In Section 3
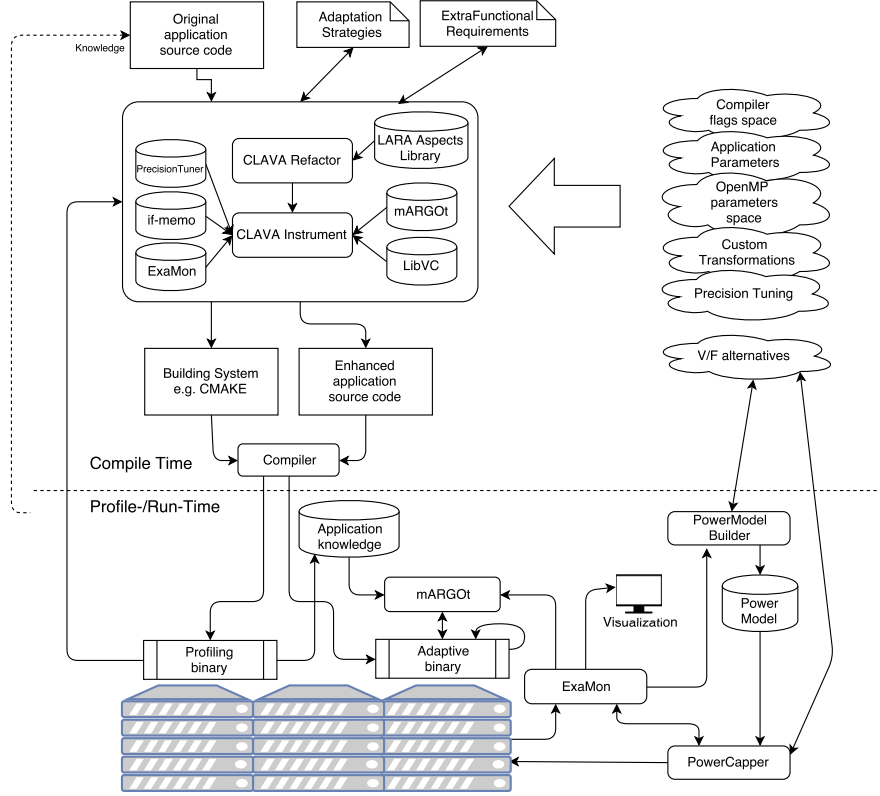
Figure 1: The ANTAREX Tool Flow, comprised of a compile-time part, where the instrumentation of the functional code is performed based on the strategies expressed in the ANTAREX DSL, and a run-time part, where the generated adaptive binary is monitored during its run on the system and optimised by tuning its parameters, including selection of different code versions.

we provide an assessment of the impact of the proposed DSL on application specifications, while Sections 4 and 5 show of how the Tool Flow has been applied the two use cases of the project. Finally, in Section 6 we compare the ANTAREX DSL with related works from the recent literature, and in Section 7 we draw some conclusions.

2. ANTAREX Technology Portfolio

In this Section, we first provide an overview of the ANTAREX DSL itself, and then we show how the DSL is used to apply each of the main components of the Tool Flow, providing examples of the code needed in each case to allow the reader to understand the level of complexity involved.

2.1. 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

4

(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., [6]) 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) [7], mainly specifying adaptation concerns, targeting specific execution points, separately from the primary functionality of the application, with minimum or no changes to the application source code.

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 main functionality. Those requirements are expressed as DSL aspects that embody strategies. An extra compilation step, performed by a *weaver*, merges the original source code and aspects into the intended program [8]. Using aspects to separate concerns from the core objective of the program can result in cleaner programs and increased productivity (e.g., reusability of strategies). As the development process of HPC applications typically involves two types of experts (application-domain experts and HPC system architects) that split their responsibilities along the boundary of functional description and extra-functional aspects, our DSL-aided toolflow provides a suitable approach for 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*[4] 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., [9]), 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 [10] or compiler optimization sequences [11]. 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[5] 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.

2.2. Precision Tuning

Error-tolerating applications are increasingly common in the emerging field of real-time HPC, allowing to trade-off precision for performance and/or energy. Thus, recent

---

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

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

5

works investigated the use of customized precision in HPC as a way to provide a breakthrough in power and performance. We developed a set of LARA aspects enabling mixed precision tuning on C/C++ and OpenCL kernels. In our precision tuning we combine an adaptive selection of floating and fixed point arithmetic, targeting HPC applications.

Figure 2 presents part of a LARA strategy that changes all declarations of a certain type to a target type (e.g., from double to float) for a given function. We note, however, that a practical and reusable aspect needs to deal with further issues, such as the cloning of functions whose types we want to change but are also called by other unrelated functions in the code, assignments of constants, casts, recursion, changing functions definitions and library functions to the ones related to the type used (e.g., `sqrtf` vs `sqrt` in Math.h), etc. In this example, `changeType` is a function that analyzes and changes compound types, such as `double*` and `double[]`. If the type described in `$old` is found inside the type of the declaration, it is replaced with the type described in `$new`. To be more specific, if `$old` is `double`, `$new` is `float` and `$decl.type` is `double*`, the type of the declaration will be changed to `float*`. If the original declaration type does not contain the `$old` type, it is not changed.

```
1 aspectdef ChangePrecision
2
3   input $func, $old, $new end
4
5   /* change type of variable declarations found
6    *  inside the function and parameters */
7   select $func.decl end
8   apply
9     var changedType = changeType($decl.type, $old, $new);
10    def type = changedType;
11  end
12
13  /* do the same with the function return type ... */
14  var $returnType = $func.functionType.returnType;
15  $func.functionType.def returnType =
16        changeType($returnType, $old, $new);
17 end
```

Figure 2: Example of LARA aspect to change the types of variables declared inside a given function.

A LARA aspect consists of three main steps. Firstly, one captures the points of interest in the code using a `select` statement, which in this example selects variable declarations. Then, using the `apply` statement, one acts over the selected program points. In this case, it will define the types of the captured declared variables, using the `type` attribute. Finally, we can then specify conditions to constrain the execution of the `apply` (i.e., only if the declared variable has a specific type). This can be done via conditional statements (`if`s) as well as via special `condition` blocks that constrain the entire `apply`. LARA promotes modularity and aspect reuse, and supports embedding JavaScript code, to specify more sophisticated strategies. As shown in [12], we support exploration of mixed precision OpenCL kernels by using half, single, and double precision floating point data types. We additionally support fixed point representations through a custom C++ template-based implementation for HPC systems,

which has already been used in [13]. In both cases the LARA aspects automatically insert code for proper type conversion before and after the critical section that has been converted to exploit a reduced precision data type.

The LARA aspect in Figure 3 shows the generation of different mixed-precision versions to be dynamically evaluated. It is possible to specify – as input of the aspect – the number of mix combination to generate, and a rule set to filter out precision mix combinations which are very likely to lead to useless and/or not efficient results. We exploit programmer's application domain knowledge by relying on them to define test cases to evaluate the different code versions at runtime. LARA automatically inserts code to dynamically perform the exploration over the space of the generated versions with different precision mix.

```
1  aspectdef HalfPrecisionOpenCL
2    input
3      combinationFilter = [],
4      maxVersions = undefined
5    end
6
7    // List of float and double vars in the OpenCL kernel
8    call result : OpenCLVariablesToTest;
9    var variablesToTest = result.variablesToTest;
10   // Sequence generator
11   var sequenceGenerator = new SequentialCombinations(
12     variablesToTest.length, maxVersions);
13   var counter = 0;
14   while(sequenceGenerator.hasNext()) {
15     Clava.pushAst(); // Save current AST
16
17     // Get a new combination of variables
18     var combination = sequenceGenerator.next();
19     var lastSeed = sequenceGenerator.getLastSeed();
20     if(!isCombinationValid(lastSeed, combinationFilter))
21       continue;
22     // Change the builtin type of the variables
23     for(var index of combination) {
24       var $vardecl = Clava.findJp(variablesToTest[index]);
25       changeTypeToHalf($vardecl);
26     }
27     call addHalfPragma(); // Enable half-precision
28     var outputFolder = createFolder(lastSeed,
29       variablesToTest.length, counter);
30     Clava.writeCode(outputFolder); // Generate code
31     Clava.popAst(); // Restore previous AST tree
32     counter++; // Increase counter
33   }
34 end
```

Figure 3: Example of LARA aspect that generates different precision mix versions of the same OpenCL kernel.

2.3. Code Versioning

One of the strategies supported in the ANTAREX toolflow is the capability to generate versions of a function and to select the one that satisfies certain requirements at runtime. Figure 4 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.

```
1  import ChangePrecision;
2  import clava.ClavaJoinPoints;
3
4  aspectdef CreateFloatVersion
5    input $func, suffix end
6    output $clonedFunc end
7
8    $double = ClavaJoinPoints.builtinType('double');
9    $float = ClavaJoinPoints.builtinType('float');
10
11   /* clone the target functions and the child calls */
12   var clonedFuncs = {};
13   call cloned : CloneFunction($func, suffix, clonedFuncs);
14
15   /* change the precision of the cloned function */
16   for($clonedFunc of clonedFuncs)
17     call ChangePrecision($clonedFunc, $double, $float);
18
19   $clonedFunc = cloned.$clonedFunc;
20  end
```

Figure 4: Example of LARA aspect to clone an existing function and change the type of the clone.

The aspect `Multiversion` – in Figure 5 – 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 4 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 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 6.

In the ANTAREX toolflow, the capability of providing several versions of the same function is not limited to static features. LIBVERSIONINGCOMPILER [14, 15] (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

8

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

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 [16, 17]. 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 7 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

```
1  switch (version) {
2    case 0: {
3      clock_gettime(CLOCK_MONOTONIC, &time_start_0);
4      SumOfInternalDistances(atoms, 1000);
5      clock_gettime(CLOCK_MONOTONIC, &time_end_0);
6      double time_0 = calc_time(time_start_0, time_end_0);
7      printf("Original time:%fms\n", time_0);
8    }
9    break;
10   case 1: {
11     clock_gettime(CLOCK_MONOTONIC, &time_start_1);
12     SumOfInternalDistances_f(atoms_f, 1000);
13     clock_gettime(CLOCK_MONOTONIC, &time_end_1);
14     double time_1 = calc_time(time_start_1, time_end_1);
15     printf("Float time::%fms\n", time_1);
16   }
17   break;
18 }
```

Figure 6: Excerpt of the C code resulting from the generation of alternative code versions.

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.

```
1  import antarex.libvc.LibVC;
2
3  aspectdef SimpleLibVC
4
5    input
6      name, $target, options
7    end
8
9    var $function = $target.definition;
10   var lvc = new LibVC($function, {logFile:"log.txt"}, name);
11
12   var lvcOptions = new LibVCOptions();
13   for (var o of options) {
14     lvcOptions.addOptionLiteral(o.name, o.value, o.value);
15   }
16   lvc.setOptions(lvcOptions);
17
18   lvc.setErrorStrategyExit();
19
20   lvc.replaceCall($target);
21 end
```

Figure 7: Example of LARA aspect to replace a function call to a kernel with a call to a dynamically generated version of that 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 [18, 19].

2.4. Memoization

Memoization is a technique that trades off computation time for memory space by storing computed values and reusing them instead of recomputing them. It has been shown to be effective in improving performance and reducing energy consumption in large scale applications [20]. We introduce in this section a memoization technique integrated in the ANTAREX toolflow. Performance can be improved by caching results of pure functions (i.e. deterministic functions without side effects), and retrieving them instead of recomputing a result. We have implemented the work of [21] generalized for C++ and aided with extensions regarding user/developer flexibility. We describe here only the principles of this technique and more details can be found in [22] [21].

```
1       float foo (float p) {
2       /* code of foo without side effects */
3       }
4
5       float foo_wrapper(float p)
6           {
7           float r;
8           /* already in the table ? */
9           if (lookup_table(p, &r)) return r;
10          /* calling the original function */
11          r = foo(p);
12          /* updating the table or not */
13          update_table(p, r);
14          return r;
```

Figure 8: A memoizable C function and its wrapper.

Consider a memoizable C function `foo` as shown in Figure 8. The memoization consists in:

1. the insertion of a wrapper function `foo_wrapper` and an associated table.

2. The substitution of the references to `foo` by `foo_wrapper` in the application.

This technique has been extended for C++ memoizable methods and takes into account the mangling, the overloading, and the references to the objects. Memoization is 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.

An example of a LARA aspect for memoization is shown in Figure 9. It defines the memoization (lines 1-13) of a method (`aMethod`) of a class (`aClass`) with `nbArg` parameters of same type as the returned type (`Type`). Note that the inputs `nbArg` and `Type` are required to manage the overloading of the object-oriented languages such that C++. Other parameters (from line 4) are provided to improve several memoization approaches. For examples, the user can specify (1) the policy in case of conflicts regarding the same table entry (line 11): replacement or not in case of conflict to the same entry of the table for different parameters of the memoized function, and (2) the size of the table (line 15). After some verifications, not detailed here, on the parameters

11

```
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 for a fully offline strategy
9   FileToSave,  // filemane to save the table, or 'none'
10  Replace,     // 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 9: An example of LARA aspect defined for the memoization.

(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 for calling the created "wrapper", this wrapper 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. To be complete about the memoization, a LARA aspect is proposed to automatically detect the memoizable functions or methods. Then the user may decide to apply or not the memoization on these selected elements.

2.5. 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 [6] [23] 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 [24]. 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 10 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 [25]. 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.

2.6. Monitoring

Today 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 privilege 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[26] (*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 value 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 called every time a new message is received. To let LARA take advantage of this monitoring mechanism we have designed the Collector API, which allow the

---

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

```
1  aspectdef mARGOt_Aspect
2  /* Input: TargetFunctionCall*/
3  input targetCallName end
4  /* mARGOt configuration */
5    var config = new MargotConfig();
6    var targetBlock = config.newBlock($targetCallName);
7    targetBlock.addKnob('Knob1', 'knob1', 'int');
8    targetBlock.addMetric('error', 'float');
9    targetBlock.addMetric('throughput', 'float');
10   targetBlock.addMetricGoal('my_error_goal',
11     MargotCFun.LE, 0.03, 'error');
12
13   /* optimization problem */
14   var problem = targetBlock.newState('defaultState');
15   problem.maximizeMetric('throughput');
16   problem.subjectTo('my_error_goal');
17
18   /* generate the information needed
19      for enhancing the application code */
20   codegen = MargotCodeGen.fromConfig(config, $targetCallName);
21
22 /* Target function call identification */
23 select stmt.call{targetName} end
24
25 /* Add mARGOt calls*/
26 codegen.init($call);
27 codegen.update($call);
28 end
```

Figure 10: Example of a LARA aspect for autotuner configuration and code enhancement.

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 11 shows a usage example of ExaMon through LARA, which subscribes to a topic on a given broker and inserts a logging message in the application. To define the connection information, the user needs 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 needs 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.

2.7. 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 [27]. 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

14

```lara
1  import antarex.examon.Examon;
2  import lara.code.Logger;
3
4  aspectdef SimpleExamon
5
6    input
7      name, ip, topic, $manageFunction, $targetStmt
8    end
9
10   var broker = new ExamonBroker(ip);
11
12   var exa = new ExamonCollector(name, topic);
13
14   // manage the collector on the target function
15   select $manageFunction.body end
16   apply
17     exa.init(broker, $body);
18     exa.start($body);
19
20     exa.end($body);
21     exa.clean($body);
22   end
23
24   // get the value and use it in the target stmt
25   exa.get($targetStmt);
26
27   // get the last stmt of the scope of the target stmt
28   var $lastStmt = $targetStmt.ancestor("scope").lastStmt;
29   // Create printf for time and data
30   var logger = new Logger();
31   logger.ln().text("Time=").double(getTimeExpr(exa))
32     .text("[s], data=").double(exa.getMean()).ln();
33   // Add printf after last stmt
34   logger.log($lastStmt);
35 end
```

Figure 11: Example of a LARA aspect integrate an ExaMon collector into an application.

be more energy efficient than the ones selected by RAPL while still respecting the same power budget [28]. 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 [29, 30]. In ANTAREX, these APIs can be inserted by LARA aspects in the application code.

3. 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.

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

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 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 underestimate 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.

4. Case Study: Computer Accelerated Drug Discovery

In this section, we provide an overview of the application of the ANTAREX tool flow to the Drug Discovery application developed in Use Case 1.

Computational discovery of new drugs is a compute-intensive task that is critical to explore the huge space of chemicals with potential applicability as pharmaceutical drugs. Typical problems include the prediction of properties of protein-ligand complexes (such as docking and affinity) and the verification of synthetic feasibility.

In the ANTAREX project it is expected that this application supports the exploration of millions of ligands over several pockets with sizes ranging from a few thousand atoms to several tens of thousands, in a timely manner. Performance is a key factor for this application, and we have used the ANTAREX tool flow to optimize the use case for Exascale systems.

4.1. Auto-parallelization and exploration through the ANTAREX DSL

Experiments on using an entire cluster for execution of the Use Case 1 have shown that relying only on MPI to parallelize the code both to distribute parallelism between nodes and to take advantage of parallelism inside a node is highly inadequate. We observed that a few tens of thousand tasks are enough to make MPI fail during execution. To overcome that, we opted to use MPI only for distributing work between nodes, and parallelize the application inside the node with OpenMP.

We have developed an auto-parallelization LARA library that uses static analysis to detect and insert OpenMP pragmas in `for` loops that can be parallelized. We have identified the function `MeasureOverlap` in the UC1 application as an important hotspot, and we have extracted it to a test application written in C (the auto-parallelization library currently only supports C).

Figure 12 shows the LARA code that parallelizes the application. Line 6 calls the parallelization library, which parallelizes all the `for` loops in the code that where detected as safe to be parallelized. However, this can be a very inefficient strategy, since nesting parallelism with OpenMP can lead to excessive overhead.

The code in lines 9-28 detects and disables all nested OpenMP pragmas. It starts by iterating over all OpenMP pragmas in the code of the kind `parallel for`; then, it retrieves all descendants of the loop corresponding to that pragma, that are also OpenMP pragmas of the kind `parallel for`, and transforms those nested pragmas into inlined comments.

After parallelizing the code, we want to explore two parameters: the number of threads to use, and the size of the pocket to test. Figure 13 shows the LARA code for a strategy that uses the LARA library LAT[7] to explore these parameters. The strategy sets

---

```
1  import clava.autopar.Parallelize;
2
3  aspectdef ParallelizeOuterPragmas
4
5    // Parallelize all loops
6    Parallelize.forLoops();
7
8    // Disable nested OpenMP pragmas
9    select omp end
10   apply
11     if($omp.kind !== 'parallel for') {
12       continue;
13     }
14
15     if($omp.target === undefined) {
16       continue;
17     }
18
19     // 'target' is the statement associated with the pragma
20     // (a 'for' loop in this case)
21     for($innerOmp of $omp.target.descendants('omp')) {
22       if($innerOmp.kind !== 'parallel for') {
23         continue;
24       }
25       // Transform the inner OpenMP pragma into an inlined comment
26       $innerOmp.replaceWith('// ' + $innerOmp.code);
27     }
28   end
29 end
```

Figure 12: LARA code for loop parallelization.

the number of threads and the number of atoms of the pocket size to explore, and creates versions of the code for all combinations of these parameters. For each combination, the strategy will compile, run and collect the results, which are aggregated into a CSV file at the end.

In more detail, line 10 creates an instance of the Lat class, which will be used throughout the strategy. Since the code has been instrumented with OpenMP pragmas, line 13 sets some necessary compilation flags. Line 16 select the section of code we want to explore (i.e., the call to the function MeasureOverlap), and lines 17-20 set the call as the place in the code where we want to measure the metrics, and the body of the function where the call appears as the place where the parameters will be changed.

Lines 26-28 create the ranges of values we want to explore: a set of predefined values for the variable num_pocket_atoms, and a set of number of threads that start at 1 and double at each step.

Lines 33-34 set the metrics we want to measure (i.e., execution time and energy), line 37 sets the number of times we should repeat the measures for each version of the code, and line 38 executes the exploration. Finally, line 39 collects the results to a CSV file.

4.2. Performance

We have applied the auto-parallelization and exploration strategy to the C version of the MeasureOverlap function, considering pocket sizes from 5.000 atoms to

18

```
1  import lat.Lat;
2  import lat.LatUtils;
3  import lat.vars.LatVarOmpThreads;
4
5  import lara.metrics.EnergyMetric;
6  import lara.metrics.ExecutionTimeMetric;
7
8  aspectdef UC1Exploration
9
10    var lat = new Lat('uc1_exploration');
11
12    // Compiler flags
13    lat.loadCmaker().cmaker.addFlags('-fopenmp', '-O3', '-march=native');
14
15    // Scope for variable change
16    select function{"main"}.body.call{"MeasureOverlap"} end
17    apply
18      lat.setMeasure($call);
19      lat.setScope($body);
20    end
21
22    // Design space
23    pocketSizes = [5000, 7000, 10000, 12000, 50000];
24    maxThreads = 5;
25
26    var samples = new LatVarList("num_pocket_atoms", pocketSizes);
27    var threadValues = new LatVarRange("numThreads", 0, maxThreads, 1,
28                                       function(x){return Math.pow(2, x);});
29    var numThreads = new LatVarOmpThreads(threadValues);
30
31    lat.addSearchGroup([samples, numThreads]);
32
33    // Metrics
34    lat.addMetric(new ExecutionTimeMetric());
35    lat.addMetric(new EnergyMetric());
36
37    // Tune
38    lat.setNumTests(5);
39    var results = lat.tune();
40    LatUtils.resultToCsv(results, './');
41  end
```

Figure 13: LARA code for parameter exploration.

50.000 atoms. We run the experiments on a computer with an Intel Xeon CPU E5-1650 v4 processor, at 3.60GHz and 64GB of RAM. The code was compiled with gcc 7.3.0.

Figure 14 shows the improvements in execution time (i.e., speedup) and energy (i.e., energy improvement) when the number of threads changes, when considering the biggest pocket size (15.000 atoms). We use as input the sizes of 120k ligands, based on a database provided by the industrial partner.

As expected for this case, the improvement curve of the execution time coincides with the improvement in energy consumed, and increases with the number of threads. The number of atoms in the pocket size modestly changes the speedup, i.e. between 0.7 % and 5.5 %, for the same number of threads. The amount of change increases with the number of threads, but for a given number of threads, an increase in pocket size improves speedup up to a point, and after that the speedup decreases (we consider

that this happens due to saturation of memory resources). This kind of exploration can be used to generate data which can be fed to the autotuner.
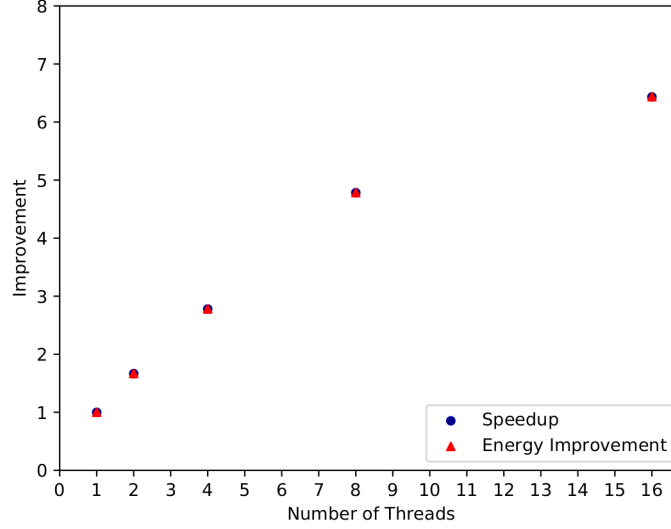

Figure 14: Results of the DSL-enabled automatic exploration

5. Case Study: 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.

5.1. Memoization and Precision Tuning through the ANTAREX DSL

The navigation system in our use case calculates routes simultaneously for possibly all participants in the system and it much relies on precise calculation of the macroscopic traffic flow model. The critical component for our traffic flow calculation is the

Betweenness Centrality algorithm, which also poses the biggest challenge for computation efficiency. The Betweenness Centrality algorithm can alternatively be used as a tool for a what-if analysis on a real traffic network [31].

We have applied two source-to-source transformations to the Betweenness Centrality algorithm: custom precision and memoization. The original application contains 770 lines of the code (C code + the header files). The LARA aspect in Figure 15 shows the main strategy for the custom precision transformation. It replaces the references to `double` type by a new symbol and replaces all the syntactic expressions that depend on the `double` type (for instance, mathematical functions) by new symbols. The custom precision library generates several files that allow to have different definitions for the created symbols: a file that defines the created parameter symbols by the original values (`double` here); a file that defines the created symbols to have a `float` version, generated by the call to the `CustomPrecisionGenParametersFor('float')` aspect; a file that contains a template to have another type representation. The last one must be updated by the user to a specific type representation (for example: half precision, fixed point representation, etc.). After applying the custom precision strategy, the final number of lines of the Betweenness Centrality application is equal to 870.

```
1  import clava.Clava;
2  import clava.ClavaJoinPoints;
3  import antarex.utils.messages;
4  import antarex.precision.CustomPrecision;
5
6  aspectdef Launcher
7    println('ANTAREX UC2, Custom precision')
8    call CustomPrecision_Initialize();
9    call CustomPrecisionFor('double');
10   call CustomPrecisionGenParametersFor('float');
11   call CustomPrecision_Finalize();
12 end
```

Figure 15: Custom precision LARA launcher for Betweenness application

For the memoization strategy, which was used as a proof-of-concept for this technology, the main DSL aspect is shown in Figure 16. It applies the memoization to the method `computeMetric` of the `Betweenness` class.

```
1  import clava.Clava;
2  import clava.ClavaJoinPoints;
3  import antarex.utils.messages;
4  import antarex.memoi.Memoization;
5
6  aspectdef Launcher
7    println('ANTAREX UC2, Memoization')
8    call Memoize_Initialize();
9    call Memoize_Method('Betweenness', 'computeMetric');
10   call Memoize_Finalize();
11 end
```

Figure 16: Lara launcher for memoization in Betweenness application

Table 3: Characterization of the traffic network graphs used in the experiments

Graph	Vertices	Edges
Porto city area	7,316	14,793
4EU countries	3,567,735	8,610,752

After applying the strategy, the application has 775 lines of code, and uses a custom memoization library automatically generated by Clava with around 150 lines of C code. The number of generated lines depends on the number of memoized functions (one in this case). So, the total number of lines is 925, considering the memoization library. In order to optimize the memory allocation, we applied another modification to the Betweenness Centrality algorithm. It consists of moving statements outside the body of a loop without affecting the functionality of the program (also called code hoisting). This transformation was applied manually and is not defined in LARA. We also applied the memoization technique to the routing algorithms for one function, with a similar number of generated lines as the Betweenness Centrality algorithm.

5.1.1. Performance and scalability

For testing impact of INRIA tools on Betweenness Centrality algorithm, we chose a graph of the traffic network of *Porto* city in Portugal and one bigger graph of 4 countries (*4EU* - Czech Republic, Slovakia, Austria, Hungary). This experiment was focused on comparing the speed of an original and of a modified version of the Betweenness Centrality algorithm. The size of both graphs in terms of vertex and edge count is presented in Table 5.1.1.

The performance was tested on Salomon cluster operated by the IT4Innovations National Supercomputing Center. The cluster consists of 1,008 compute nodes and each of them contains 2x Intel Xeon E5-2680v3 processors clocked at 2.5 GHz and 128 GB DDR4@2133 RAM. When the experiments were performed, the operating system was CentOS 7.6 and the code was compiled using Intel C++ Compiler 17.0.

We tested several versions of the Betweenness Centrality algorithm with various combinations of the ANTAREX tools:

Float (F) a version of the algortihm with reduced precision of internal variables

Double (D) a standard algorithm

Float_Hoisting (FH) a float version with hoisting

Double_Hoisting (DH) a standard version with hoisting

Float_Hoisting_Memoization (FHM) a float version with hoisting and memoization

Double_Hoisting_Memoization (DHM) a standard version with hoisting and memoization

Table 5.1.1 shows run times with the optimizations provided by different ANTAREX tools for different number of nodes.

Table 4: Betweenness Centrality algorithm run times for the Porto graph (in seconds)

Compute Nodes	F	FH	FHM	D	DH	DHM
1	1.20	1.15	*1.12*	1.34	1.27	1.25
2	0.66	0.65	*0.64*	0.75	0.72	0.71
4	*0.41*	0.42	*0.41*	0.47	0.45	0.44

Table 5: Betweenness Centrality algorithm run times for the 4EU graph (in 10^3 seconds)

Compute Nodes	F	FH	FHM	D	DH	DHM
1	294.575	283.534	*282.233*	340.523	325.428	315.872
2	153.296	150.123	*147.966*	175.940	170.587	167.524
4	79.484	79.233	*75.314*	89.905	87.206	86.106
8	40.811	39.899	*38.711*	46.031	44.736	44.258
16	21.305	21.070	*20.247*	23.725	22.958	22.493
32	11.122	10.809	*10.590*	12.228	11.884	11.791
64	5.922	5.726	*5.677*	6.489	6.284	6.092

Overall, we can see a speedup of 3.7 to 7.8% for the larger graph when adding the hoisting and memoization transformations to the double precision floating point code. Considering also the precision reduction, the performance speedup grows to 14.3 to 20.6% depending on the number of cores employed.

5.2. Autotuning for Client-side navigation

The Navigation application is the one which finally sells the whole navigation system, so it requires quite an attention. In a holistic approach the application implementation needs to consider multiple aspects, both in the functional and the performance space as was stipulated in the specification of the system. In simple words:

- Navigation shall not exceed defined data consumption

- Navigation shall contribute with car data to build the knowledge of the server-side system

- Navigation secures the required navigation quality

- Navigation optimizes communication with server-system to alleviate it from over loading

All these goals have been addressed through the integration of a run-time autotuner.

Important from the HPC perspective is that a navigation client acts as an intelligent source of computational demand, able to regulate the number of requests based on the quality of the navigation and the ability to perform local route computation.

Figure 5.2 shows the tool-flow for configuring the autotuner towards statistically the best performance. The input is the data consumption statistics on estimated data transfer per request type for all service modules, while collected from prior test runs (circa 100 hours) of the navigation app under prevailing commute conditions (in Bratislava
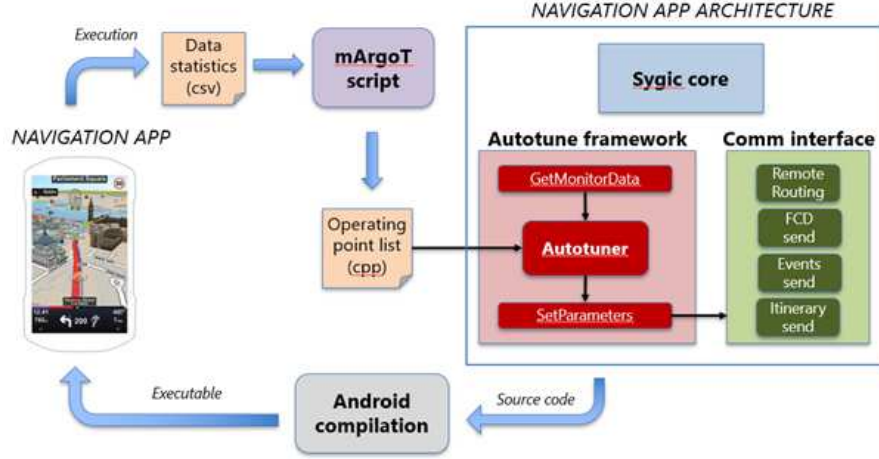
Figure 17: Autotuning framework application to the navigation app

city). As the autotuner architecture evolved the definition of the input has been, in addition to mean, extended to contain the mean, max, min, and the standard deviation of the variables.

Figure 5.2 shows the effectiveness of the mArgot autotuner compared with the baseline autotuner included in the commercial release of Sygic Truck Navigation v13.7.0 (March 2017). The baseline (red dots) is a simple autotuning implementation that supports only a data transfer limit set at 20MB. It does not support constraints on the navigation quality, providing the maximum quality based on its prediction of data usage.

The mArgot autotuner (green dots) instead takes into account a *Navigation Quality Index* (NQI) as a constraint. The NQI is defined as a fuction of the remote routing frequency, which saturates at a point dependent on the current traffic level. The constraint in the proposed experiment is for the NQI to stay above a minimum of 6.0. Furthermore, it uses the same data transfer limit (20MB), a minimal data contribution of 20%, and an estimate of 40 monthly driving hours.

As shown in Figure 5.2 the objectives are well met with an exception of few dots about ideal consumption rate, which can be explained with stochasticity of driver behaviour records (data generated only in the case of excessive acceleration, braking, cornering).

The impact of the NQI constraint selection can be understood through the experiments reported in Figure 5.2. In the experiment, the NQI is set to different values in the range [6.0,9.0]. The results show how relaxing the quality level allows to save computational resources, lowering the total cost of routing, e.g., by 12% when dropping from NQI 8.0 to 7.0. The results also allow to identify the best trade-off from a value-at-cost solution in practical cases, at NQI 6.8. Compared to the baseline autotuning, which achieves an NQI of 6.25, the mArgot autotuner allows a saving of 14% in
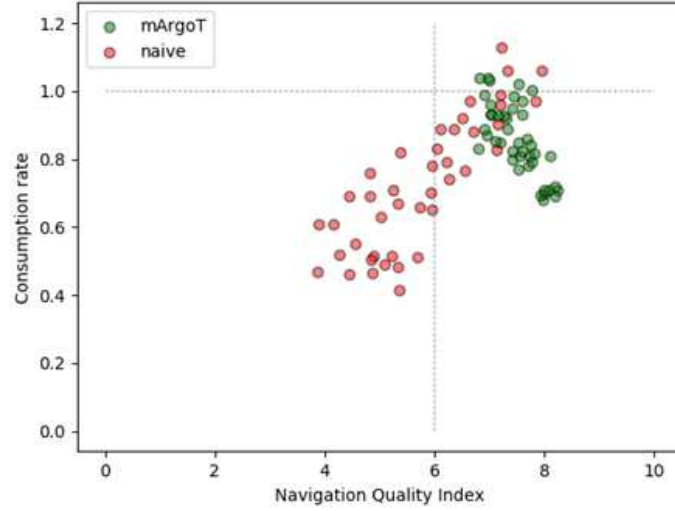
24

Figure 18: Performance comparison between baseline and mArgot autotuning

computational resources while providing a better quality of service, assuming the use of the best trade-off NQI of 6.8.

6. Related Works

Many Domain Specific Languages have been proposed for the domain of High Performance computing [32]. A survey of tools and approach in generative programming for high performance computing can be found in [33], whereas a collection of techniques for and examples of modern DSL design can be found in [34].

Many of them actually focus on a specific class of HPC applications, either in terms of a specific field such as computational biology [35], or a specific mathematical problem, such as partial differential equations [36, 37, 38, 39], often using stencil computation, i.e., common functionalities in a given field implemented in an optimised way and offered through the DSL to the user, possibly with some autotuning capabilities. This approach is effective in terms of combining efficiency of the generated code with familiarity of the language for the domain expert. However, a different solution is needed for each domain or problem, and when HPC experts at a supercomputing center are called for to optimise a customer's code, this forces specialisation, which may be undesirable from the point of view of the supercomputing center management.

Indeed, at least some such centers have attempted to develop a coherent set of languages and tools applicable across multiple application domain within the wider context of HPC [40], and large scale research programs have been devoted to this goal[41]. These DSLs are also related to algorithmic skeleton frameworks [42], which however do not take into account the presence of the HPC expert. Model-driven approach have been also proposed [43, 44], with the goal to abstract the application from the concrete
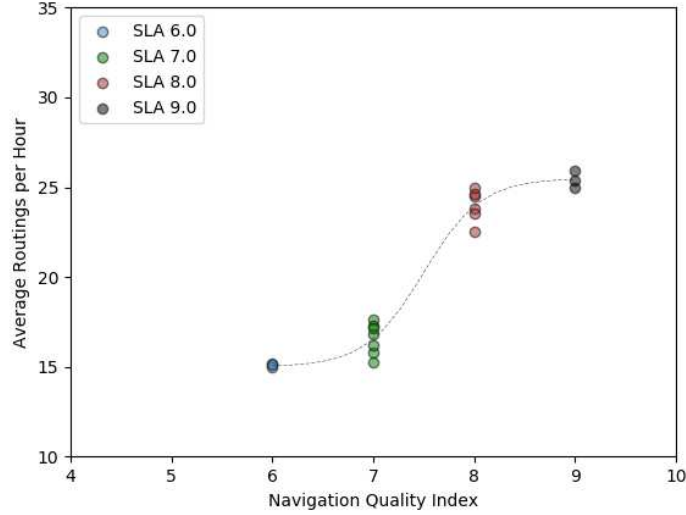
25

Figure 19: Variation of the compute workload (in routing requests per hour) as a function of the Navigation Quality Index for a scenario of medium traffic commute in Bratislava

and ever-changing HPC infrastructure. A more general-purpose take on the stencil computation is Terra [45], which employs Lua to support the creation of DSL compilers which can interoperate with the generated code.

All these efforts have the opposite problem – they force the application developer to acquire skills typical of the HPC experts. Furthermore, they do not deal easily with the large amount of legacy code found in HPC, where applications may remain in use for decades [32, 44]. The ANTAREX DSL, on the other hand, is designed exactly to cater to these needs. By creating an external set of LARA aspects, the ANTAREX DSL strategies, the HPC expert can manipulate the code developed by the application domain expert. The DSL code can be as small or as large as needed to reach the performance goals, but never intrudes in the ability of the application developers to maintain or extend their applications. At the same time, by providing access to commonly used computation patterns, code transformations, and autotuning tools, the ANTAREX DSL enables the HPC experts to reuse solutions across different applications, removing the need to reinvent the wheel with every new application domain.

7. Conclusions

In this paper, we have reported the major concrete outcome of the ANTAREX research project, the ANTAREX DSL. The DSL, based on the AOP language LARA, aims at supporting a well-established collaboration model between users and support staff of HPC centers, where the users develop applications focusing on the functional requirements, and then take advantage of the support staff's expertise on HPC to improve the performance of the applications. With the ANTAREX DSL, this process

is simplified by removing the need for the HPC support staff to manually modify the user's code, instead relying on aspects which encode available code transformations as well as providing access to libraries for autotuning, system monitoring, and dynamic recompilation.

We demonstrate the use of our tools in a scenario where the HPC center runs remote navigation for a pool of users, considering both urban and transnational scenarios, as well as in a scenario where the HPC center runs a drug discovery appplication.

Acknowledgements

[1] J. Curley, "HPC and Big Data," *Innovation*, vol. 12, no. 3, Jul. 2014.

[2] C. Silvano, A. Bartolini, A. Beccari, C. Manelfi, C. Cavazzoni, D. Gadioli, E. Rohou, G. Palermo, G. Agosta, J. Martinovič *et al.*, "The ANTAREX Tool Flow for Monitoring and Autotuning Energy Efficient HPC Systems (Invited paper)," in *SAMOS 2017-Int'l Conf on Embedded Computer Systems: Architecture, Modeling and Simulation*, 2017.

[3] C. Silvano, G. Agosta, A. Bartolini, A. R. Beccari, L. Benini, L. Besnard, J. Bispo, R. Cmar, J. M. P. Cardoso, C. Cavazzoni, S. Cherubin, D. Gadioli, M. Golasowski, I. Lasri, J. Martinovic, G. Palermo, P. Pinto, E. Rohou, N. Sanna, K. Slaninova, and E. Vitali, "Antarex: A dsl-based approach to adaptively optimizing and enforcing extra-functional properties in high performance computing," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Aug 2018, pp. 600–607.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] R. Nobre, L. Reis, J. Bispo, T. Carvalho, J. M. P. Cardoso, S. Cherubin, and G. Agosta, "Aspect-driven mixed-precision tuning targeting gpus," in *Proc 9th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 7th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, Jan 2018.

[13] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys, "Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error," in *Int'l Conf on Parallel Computing (ParCo)*, Sep 2017.

[14] 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.

[15] M. Festa, N. Gervasoni, S. Cherubin, and G. Agosta, "Continuous Program Optimization via Advanced Dynamic Compilation Techniques," in *Proc 10th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 8th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms (to appear)*, Jan 2019.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] G. Agosta, M. Bessi, E. Capra, and C. Francalanci, "Dynamic memoization for energy efficiency in financial applications," in *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE, 2011, pp. 1–8.

[21] A. Suresh, E. Rohou, and A. Seznec, "Compile-Time Function Memoization," in *26th International Conference on Compiler Construction*, Austin, United States, Feb. 2017.

[22] A. Suresh *et al.*, "Intercepting Functions for Memoization: A Case Study Using Transcendental Functions," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, p. 23, Jul. 2015.

[23] 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.

[24] 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

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[31] J. Hanzelka, M. Běloch, J. Křenek, J. Martinovič, and K. Slaninová, "Betweenness propagation," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2018, pp. 279–287.

[32] M. Giles and I. Reguly, "Trends in high-performance computing for engineering calculations," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2022, p. 20130319, 2014.

[33] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua, "In search of a program generator to implement generic transformations for high-performance computing," *Science of Computer Programming*, vol. 62, no. 1, pp. 25–46, 2006.

[34] M. Mernik, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments: Recent Developments*. IGI Global, 2012.

[35] J. Starruß, W. de Back, L. Brusch, and A. Deutsch, "Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology," *Bioinformatics*, vol. 30, no. 9, pp. 1331–1332, 2014.

[36] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy *et al.*, "Liszt: a domain specific language for building portable mesh-based pde solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 9.

[37] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. Kelly, "Pyop2: A high-level framework for performance-portable simulations on unstructured meshes," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1116–1123.

[38] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich, "Exaslang: a domain-specific language for highly scalable multigrid solvers," in *Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on*. IEEE, 2014, pp. 42–51.

[39] R. Membarth, F. Hannig, J. Teich, and H. Kostler, "Towards domain-specific computing for stencil codes in hpc," in *2012 SC Companion: High-Performance Computing, Networking, Storage and Analysis (SCC)*. IEEE, 2012, pp. 1133–1138.

[40] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[41] M. Weiland, "Chapel, fortress and x10: novel languages for hpc," *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706*, 2007.

[42] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[43] M. Palyart, D. Lugato, I. Ober, and J.-M. Bruel, "Mde4hpc: an approach for using model-driven engineering in high-performance computing," in *International SDL Forum*. Springer, 2011, pp. 247–261.

[44] M. Palyart, I. Ober, D. Lugato, and J.-M. Bruel, "Hpcml: a modeling language dedicated to high-performance scientific computing," in *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing*. ACM, 2012, p. 6.

[45] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 105–116.