H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

Deliverable D2.1: Programming Model and DSL for Adaptivity

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

Horizon 2020
European Union funding
for Research & Innovation

Deliverable Title:	Programming Model and DSL for Adaptivity		
Lead beneficiary:	UPORTO (Portugal)		
Keywords:	DSL, Programming Models for Adativity		
Author(s):	João M.P. Cardoso (UPORTO); João Bispo (UPORTO); Pedro Pinto (UPORTO); Jorge Barbosa (UPORTO); Rui Abreu (UPORTO); João Canas Ferreira (UPORTO); Giovanni Agosta (POLIMI); Stefano Cherubin (POLIMI)		
Reviewer(s):	Cristina Silvano (POLIMI); Gianluca Palermo (POLIMI); Davide Gadioli (POLIMI); Erven Rohou (INRIA)		
WP:	WP2	Task:	T2.1
Nature:	Other	Dissemination level:	PU
Identifier:	D2.1	Version:	V0.22
Delivery due date:	September 1st, 2016	Actual submission date:	September 15th, 2016

Executive Summary:

This report (issued at M12) is related to the Task 2.1 (Definition of Programming Model and DSL for Adaptivity) under the leadership of UPORTO. It describes the programming model to be used by the DSL to specify strategies for runtime adaptivity and compiler optimizations. The report describes the main artefacts for the DSL and the LARA extensions to support them.

This document is organized as follows:

- Section 1 introduces the report.
- Section 2 describes the software-based approaches for runtime adaptivity.
- Section 3 describes the ANTAREX adaptivity/autotuning approach.
- Section 4 describes the LARA-based cross-layer approach.
- Section 5 describes the programming model and DSL features to address the ANTAREX requirements.
- Section 6 concludes the report.

Approved and issued by the Project Coordinator: 	**Date:** September 15th, 2016

Project Coordinator: Prof. Dr. Cristina SILVANO – Politecnico di Milano
e-mail: silvano@elet.polimi.it - **Phone:** +39-02-2399-3692- **Fax:** +39-02-2399-3411

Table of Contents

1 Introduction

One main feature of the ANTAREX project is to provide to applications the capability to adapt the application at runtime according to specific requirements, such as performance, energy consumption, power dissipation, and quality of service (QoS). This adaptation crosscuts in ANTAREX various layers and must be seen in a holistic way, from application parameters, compiler optimizations, to runtime decisions.

The runtime adaptivity dimensions to be supported by the ANTAREX approach are the following:

- Modifications of **application parameters (attributes);**
- Selection among different **algorithms** for solving the same problem;
- Different **compiler optimizations** for the same algorithm;
- **Runtime strategies** for partitioning and for mapping computations targeting hardware accelerators;
- **Runtime management** of system resources.

An analysis of the adaptivity requirements - gathered from the analysis of the application requirements done in Task 1.1 and from more generic requirements elicited by our Supercomputing Center partners - has been conducted. Table I shows examples of tuning parameters (knobs) and monitoring/observation needs according to the two ANTAREX use cases and the two Supercomputing Center Requirements. The knobs considered provide innumerous opportunities and challenges in terms of application and compiler optimizations, autotuning policies, and in terms of the programming model and Domain-Specific Language (DSL) support.

This report describes the programming model and the DSL support for the ANTAREX approach. The report is organized as follows. Section 2 briefly describes the most representative work regarding the adaptivity in terms of software and compilers. Section 3 describes the ANTAREX autotuning approach in terms of the programming model and DSL support. Section 4 describes the programming model and the DSL. Section 5 briefly describes the work to be done in the context of REFLECT. Section 6 concludes the report.

Table I. Representative examples of knobs and monitoring information for adaptivity and autotuning.

Stack Level	**Tuning Parameters/Knobs (act)**	**Monitor/Observe**
Software (application level)	Different algorithms for the same function (e.g., possibly considering approximate computing options); Different implementations of a function (algorithm) (including specialization, approximate computing); Input parameters of the application (such as the threshold value for approximating distances);	Parameters of the application (e.g., size of an array); Function to measure QoS, accuracy; Values of function arguments (e.g., arguments of calls to pure functions), range values;
Compiler Optimizations (including source to source)	Loop tilling block size; Loop perforation skip parameter; Selection of multiversioning functions; Memoization options; Use of built-in functions (e.g., memalign) and their parameters; Allocation of application threads/processes to match cache usage (CPUs); Leverage the GPGPU strategies mentioned on Subsection 3.1 (these basically require rewriting the application to some extent in order to move data from global to shared memory); Parallelization via OpenMP: • Number of threads (e.g., using built-in function) • Scheduling algorithm • Vectorization Parallelization via MPI: • Number of processes Runtime invocation of the compiler with different flags; OpenCL Knobs: • Flag options and phase ordering during offline kernel compilation; • Selection of the NDRange configuration;	Loop trip count (statically or dynamically known); Number of messages among nodes (MPI);
Application manager (including job scheduler)	Task mapping and scheduling (application level); Task migration (application level);	Data communication characteristics (size, rate, etc.);
Operating System	NA	Number of page faults Cache miss rates
Architecture	NA	NA
Hardware	Power gating; Clock gating; Dynamic Voltage and Frequency Scaling (DVFS);	Component level: Power; Energy;

	Power Budget (via RAPL);	Timing; Temperature; Clock Frequency; Activity (CPU, GPU);

2 Software-based Approaches for Targeting Runtime Adaptivity

Runtime adaptivity is an increasingly important concept, especially when considering the large potential of adapting and optimizing applications to specific scenarios, runtime data, workloads, and execution environments. To implement such a system, *IBM* suggests a reference model for autonomic control loops [Kephart2007], referred to as the *Monitor, Analyze, Plan, Execute and Knowledge* (*MAPE-K*) control loop. In the *MAPE-K* approach, the *monitor* component is responsible for acquiring information about the system execution (e.g., power consumption, execution time, CPU/GPU utilization, temperature), the *analyzer* component is responsible for pinpointing the probable source(s) of measurements (e.g., power) above/below the goals, the *plan* component provides operating points addressing system improvements regarding the target metric and goal, and the *executor* component implements the plan to achieve system improvements. In ANTAREX we will use compiler-based technology to automatically add support for runtime adaptivity, in order to achieve the intended requirements (e.g., performance, energy). We describe next relevant approaches considering runtime adaptivity, from applications to compiler optimizations.

2.1 Approaches for Programming Compiler Optimizations/Transformations

Programmers have relied on directive-driven programming models to partition and map their codes to heterogeneous platforms (e.g., OpenMP [Dagum1998], hiCUDA [Han2009], OpenACC [Wienke2012], and accULL [Reyes2012a] the open-source implementation of OpenACC). In this approach, programmers must be aware of the syntax of the directives and the corresponding concurrent semantics in order to derive correct code. This parallelization process is extremely labor intensive and therefore error prone. Worse, the need to retarget applications to multiple platforms, in combination with different programming environments, leads to the obfuscation of the original application's source code by the use of additional conditional compilation pre-processing directives (e.g., #ifdef). Despite the performance appeal, the mapping of applications to multicore systems is non-trivial for the average programmer, hence making the use of automated or semi-automated process an attractive approach. OpenMP is arguably the most supported standard pragma-driven compiler extension for adding multithreaded support to C/C++ applications targeting shared memory architectures. OpenMP includes a series of "#pragma" directives that specify sections of the code to be executed concurrently. OpenMP-like directive-driven programming models, such as OpenACC [OpenACC2013], have been proposed for targeting hardware accelerators, in many cases by translating pragma directives into kernel OpenCL or CUDA code and automatically generating initialization and communication code between the host and OpenCL/CUDA compatible devices [Reyes2012a][Wolfe2010].

Recent trends also include the use of DSLs (e.g., [Yi2012a]) for specifying domain-specific computations and compilers to generate code from those DSLs. Although this is an approach that may provide higher levels of performance and/or more energy efficient systems, it is orthogonal to the ANTAREX approach as the DSL being proposed in ANTAREX is a DSL to specify strategies related to non-functional concerns, such as "recipes" for parallelization via OpenMP directives, for monitoring and for providing custom profiling, and for transforming the code.

2.1.1 Directive-Driven Programming Models

Directive driven programming (DDP) models are a popular approach to achieve parallel implementations due to its easy-to-use directive-based style [Beyer2011] and to the portability that is usually obtained through a class of compilers. These models are mainly applicable to shared memory programming where parallelism is implemented with multiple threads that accesses the same memory space and are characterized by being language extensions. Compared to the low-level threading approach that requires efforts in data decomposition, thread spawning and forking, etc., DDP models allow a simpler conversion of sequential applications, into parallel versions, with a lower level of effort. However, these advantages come with some penalty in the performance obtained with direct multithreading for CPUs and OpenCL/CUDA for accelerators. The dominant programming models for accelerators are currently CUDA [Nvidia2010] and OpenCL [Khronos2010]. Both allow the programmer to improve performance by using the accelerator, but the rewriting of the code for the target architecture is at a low level. Alternatively, OpenMP allows expressing parallelism with simple directives for multicore CPUs, and OpenACC provides similar support for GPUs.

The Portland Group provides the PGI Accelerator programming model [PGI2010] for C and Fortran which enables compiler-aided and directive-based work offloading to NVIDIA GPUs and specifies a broad range of features. Although some performance is compromised, [Wienke2011] concluded that the PGI Accelerator model has a good effort-performance ratio.

OpenMP is the more widely used and best known DDP-model for shared memory programming but there are other efforts such as StarSs that goes beyond OpenMP, by providing a node level programming model that integrates smoothly with distributed memory models, such as MPI, and naturally supports heterogeneity naturally. OmpSs resulted from the merge of OpenMP and StarSs by combining the structured parallelism offered by OpenMP with the execution of parallel tasks in a non-structured way offered by StarSs [Elangovan2012].

Computing nodes have become heterogeneous with multicores and accelerators raising the need for autotuning techniques. Additionally, accelerators present more diversity through product generations [Elangovan2015] making it also relevant to autotune the kernels for the target hardware. Tunable parameters are the work-group size, number of registers used, and size of local memory reservation. In [Gray2012] optimization for autotuning are applied through a high level directive-based language and a source-to-source compiler that can generate CUDA/OpenCL code. The large optimization space of GPU kernels is explored by using loop permutation, loop unrolling, tiling, and in the specification of which loops to parallelize.

The support provided by the ANTAREX DSL will focus parallelization strategies able to add to application code the suitable directives and/or to refactor the application.

2.1.2 Languages Targeted at Multicore Heterogeneous Architectures

The programming model for heterogeneous architectures needs to enable the exploitation of hardware resources through language constructs that encode hot spot computations. OpenCL [OpenCL2008] is an open standard for the development of parallel applications on heterogeneous multi-core architectures [Khronos2011] consisting of GPUs, CPUs and DSPs. Moreover, OpenCL consists of a C-like language (OpenCL C) and API (OpenCL API) which allow programs to be organized as a host

code section and a device code section. The host code runs on a general-purpose (multi-) processor, and is in charge of control-intensive tasks, and controls the offloading of compute-intensive tasks to the device(s).

OpenCL provides general-purpose parallel programming across multiple types of processors, including CPUs, GPUs and, more recently, FPGAs [Altera2013]. Using OpenCL, an application running on the host (the CPU) may execute programs on one or more devices (e.g. multi-core CPUs and GPUs). Devices are grouped in platforms, which typically correspond to OpenCL implementations.

In OpenCL the goal of increased performance is achieved by the high utilization of target devices, including a fully exposed memory hierarchy, as well as by the variety of devices that can be available on the platform [Du2012]. This approach, however, requires the development of extensive boilerplate code (e.g., for setting up a device or transferring data to it) which substantially interferes with the expression of the algorithm. Moreover, code performance is very sensible to minimal variation in the architectural parameters, thus requiring lot of effort in tuning the application performances [Agosta2014][Paone2015]. To cope with these problems, some OpenMP-like extensions to C, C++, and Fortran have been proposed (e.g., OpenACC and OpenHMPP), but they cannot yet compete with the performance of hand-written code. Moreover, they impose a new and specialized syntax on the programmer.

2.1.3 Aspect-Oriented Programming (AOP) Approaches

A higher-level approach to the software development for heterogeneous platform is to adopt an aspect-oriented approach (AOP) [Kiczales1997] to apply user-specified rules and strategies for code transformations. The use of AOP to encapsulate parallel execution concerns has been the focus of various previous research efforts (see, e.g., [Harbulot2004]). Sobral [Sobral2006] showed how parallel execution issues can be modelled by multiple concerns, each one implemented by a specific AOP module. Cunha [Cunha2006] showed how common concurrency patterns and mechanisms are encapsulated into a library of Aspects. More recently, with the growing interest in multi-core and GPU systems, [Medeiros2013] developed an AOP library to implement programming constructs that resemble OpenMP in Java, while [Wang2010] showed how AOP can help to encapsulate GPU specific execution concerns. These works are efforts to move all details concerning the execution platform into aspect modules. Thus, they can be seen as AOP modules that specialize a given base program for a particular execution platform, making base programs more portable across platforms, since parallel execution concerns are moved into platform-specific aspect-modules that provide performance portability across parallel systems.

The use of dynamic weaving (at runtime) in the context of AOP has been proposed by various authors (see, e.g., [Popovici2002]). Although the AOP community has proposed various dynamic weaving approaches (see, e.g., [Villazon2009]), the join point model used by most approaches has been traditionally simpler than the one required by the code refactoring and runtime adaptivity strategies that are addressed by the ANTAREX project.

2.1.4 The Use of DSLs in Adaptivity Systems

Instead of using directives for specifying compiler optimizations and source-to-source transformations, the use of domain-specific languages (DSLs) constitutes a promising approach for expressing run-time adaptivity strategies. The definition of DSL proposed by [vanDeursen2000] is:

> *A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*

This definition puts the emphasis on the language's *focused* expressive power. A recent example of a DSL for runtime adaptivity strategies is proposed in [Santos2014]. It has been designed for programming runtime adaptivity in context-aware systems, especially when considering the adaptivity of application parameters. Another example of using a DSL for adaptivity control, in this case, of a multimedia application, is proposed within the BZR language [Delaval10]. The management of dynamic adaptations is considered to be part of a closed control loop, where system monitoring decides which adaptations to be performed, from previously specified contract policies.

Yet another research direction for runtime adaptation is the one that takes into account the software architecture of the system, providing a reusable infrastructure framework. An incarnation of such approach is the Rainbow framework [Garlan2004], which uses a software architecture and reusable infrastructure to support self-adaptation of software systems. Recently an approach has been proposed that not only provides repair strategies, but also is also able to detect and diagnose the root cause of the system, which leads to better accuracy when in need to decide which strategy to apply next [Casanova2013].

For DSLs that need to deal with crosscutting concerns (as is required in the ANTAREX project), a very fruitful approach is to adhere to the aspect-oriented programming (AOP) paradigm [Fabry15]. The language LARA [Cardoso13] is a good example of this kind of approach. LARA allows developers to capture non-functional requirements (NFRs) from applications in a structured way, leveraging high/low-level actions and flexible tool chain interfaces. Developers can thus benefit from retaining the original application source-code while exploiting the benefits of an automatic approach for various domain- and target-specific compilation/synthesis tools. LARA allows support to program code instrumentation strategies, to control the application of code transformations and compiler optimizations, and to effectively control different tools in a toolchain. Moreover, LARA provides developers with artifacts and with a unified view, which diminishes the efforts to program and evaluate design space exploration (DSE) schemes.

When dealing with dynamic environments, optimizations and program specializations may acquire increased impact if applied at runtime, since in this case they can exploit additional information. Runtime strategies can adapt applications to the resources in the target computing system, and may control resource allocation and task mapping to the cores of a multicore system, subject to specific requirements [Mariani2011]. Specific cases of strategies to specify mapping options and/or mapping techniques have been researched. The scripting language of [Yi2012a][Yi2012b] and the search-space description language of [Kim2015] address some of the issues of exposing source-level transformations in a controlled fashion by enabling the specification of sequences of transformations and corresponding parameters.

2.2 Approaches for Programming Autotuning Strategies

The information needed to decide about a particular code version and/or compiler optimization is difficult to acquire and/or can be unavailable at static compile time. Auto-tunable systems can provide the required adaptivity in terms of the compiler optimizations best suited to the particular execution environment.

The exploration of different compiler optimizations at runtime has been focus of several researchers. An example is Dynamic Feedback [Diniz1997], a technique to adapt computations to different execution environments. Dynamic Feedback considers alternating sampling and production phases and explores different versions of the code using each version a different optimization policy. Although avoided as much as possible due to the associated execution time overhead, the technique can also be applied when some code versions are provided at runtime through dynamic compilation. The sampling phase is responsible for the measurement of the overhead of each version in the current environment by running that version for a fixed time interval. In each production phase the system uses the version with least overhead in the previous sampling phase. The system periodically repeats sampling-production phases in order to adapt to the execution environment.

The Automated De-Coupled Adaptive Program Transformation (ADAPT) framework [Voss2000] [Voss2001a][Voss2001b] consists of the ADAPT language (AL), a DSL for programming runtime iterative compilation processes, including the monitoring points, the ADAPT compiler, and an infrastructure to collect measurements, to execute a policy, and to call a remote compiler. The ADAPT approach follows a separation of concerns where the adaptivity is expressed in a specific file (consisting of global variables, one or more “technique” sections and one coordination section) and without requiring modifications to the application logic. ADAPT overlaps optimization with execution. AL is a C-like language and was one of the first DSLs specially dedicated to program optimization policies in the context of runtime exploration of compiler optimizations. Optimization policies and expressed as AL technique sections which include optimization phase subsections. A coordination section is responsible to define how AL techniques are combined (AL includes linear search, parallel search, and exhaustive search). The language provides mechanisms to define (constraint) the intervals (code regions) where a given ADAPT technique is to be applied (e.g., innermost loops, perfect nested loops, loops where the number of iterations is known at the entry). AL also provides statements (apply_sec) to define the interface to the tool to be used to apply the optimization, and/or compiler parameters. Examples of interface include if the code variant is to be generated by a specific compiler flag. Collect statements (collect) define what to measure for each specific interval (e.g., timing). AL policies can use machine information (e.g., clock frequency, cache sizes), iteration counts for innermost and outermost loops, and other information regarding runtime input data through the use of macros. Besides the phase sections an AL technique can include default and safe sections. The ADAPT compiler is responsible to process both the application source code and the AL code and to generate a complete runtime system. For instance, the phase AL sections where the heuristics are defined are translated to finite-state machines.

The ANTAREX approach and the extensions to LARA being proposed are also inspired by the ADAPT approach. However, the ANTAREX approach proposes a more generic autotunner approach and a distinction between the DSL and the autotuner framework. In ANTAREX, the DSL is

responsible for managing the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) [Kephart2003] control loop phases by providing the interactions of the application to the autotuner and by providing to the autotuner the selection of the parameters and policies to be applied. In some contexts those policies can also be expressed by the DSL and translated to autotuner code, but in most cases we foresee the use of pre-existent autotuning policies provided by the autotuning library. The macros and compile-time constraints proposed in ADAPT have attributes and join points as LARA counterparts. LARA provides more powerful language constructs to express runtime optimization policies and thus we think it includes the required constructs to express sophisticated policies.

Proposals for algorithm adaptivity have been considered recently. A representative example is PetaBricks [Ansel09], an "implicit parallel language" for program development based on algorithmic choice. Using PetaBricks, the programmer can specify multiple algorithms for solving a problem, without the need to specify the one to be used in each particular situation (e.g., according to the dataset size). Instead, based on the set of algorithms to solve the problem and experimental testing, the compiler generates and autotunes an optimized hybrid algorithm. The hybrid algorithms are obtained by combining the multiple algorithms and using the best approach for each particular execution environment and dataset. The PetaBricks language consists of rules defining code transformations and the PetaBricks code is translated to C++ code by the Petabricks compiler. Recent work on PetaBricks consider a two-level input learning algorithm for algorithm autotuning [Ding2015]. PetaBricks is an inspiration for the ANTAREX approach and LARA will support and provide strategies for modifying the application for algorithm choice and will rely on the support of the autotuner proposed in the context of the ANTAREX.

2.3 Summary

The analysis presented in this section strongly indicate that currently and to our best knowledge there is not a single solution to target all the requirements considered in the ANTAREX project. A LARA based approach may provide a single DSL able to satisfy the ANTAREX programming model requirements without significant extensions to the language and to the way tools interplay with LARA strategies.

3 The ANTAREX Adaptivity/Autotuning Approach

The ANTAREX approach to runtime adaptivity and autotuning is based on the MAPE-K control loop concept [Kephart2007]. In ANTAREX, however, the adaptivity and autotuning concerns are to be applied to application code already available or under development, but not intertwined with those type of concerns. This is ensured by the support provided by the ANTAREX DSL and by the tools involved in the development and optimization process. The DSL and the tools will provide support for the two typical stages of the development and optimization process: testing stage and deploy stage. This support is highlighted in Figure 3.1. The testing stage include the monitoring and custom profiling needed to acquire information by executing the application with representative execution environments and datasets. This stage then provides data that can be used by DSL strategies to guide the optimizations and the autotuning policies. The DSL strategies shall be able to load profiling data and/or data from tools used in the analysis of the application and to use it to drive optimization and autotuning decisions.

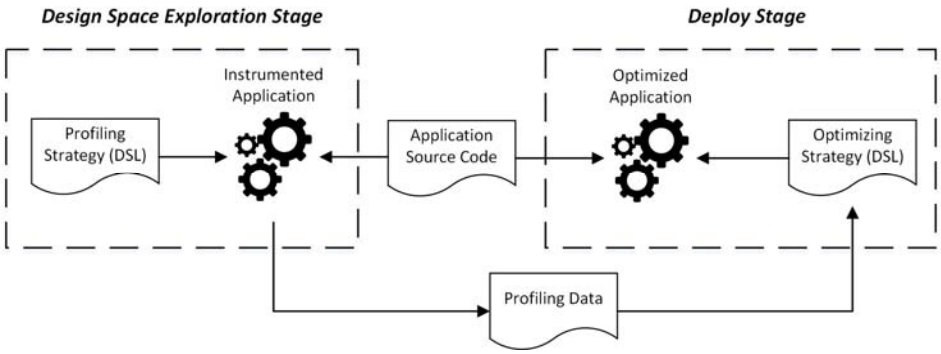

Figure 3.1 The two main ANTAREX stages. Note that the runtime adaptivity and autotuning can be part of both stages.

The main purpose of the ANTAREX Autotuner (Margo) is to provide to a code region of the input application, the most suitable configuration of the software knobs, according to the application requirement and observed environment. Usually, the managed region of code is the body of the loops that include the most demanding and tunable kernels. For this reason, the Autotuner interacts mainly in three ways with the application: to observe its actual behavior, to define/change the application requirements and to retrieve the most suitable configuration.

Figure 3.2 shows an example of a nested loop we want to target with autotuning. In this case, we want to apply "loop tiling" over the loop and then expose knobs that control the tile size. Figure 3.3 shows an example of an aspect, based on LARA, to perform loop tiling over the two innermost for loops. It receives two variables, the size of the tiling, and returns three outputs, the names of the two variables that control the size of the tile and a join point that represents the outermost of the two inner loops.

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        do_something(i, j);
    }
}
```

Figure 3.2 Example of input code

```
aspectdef LoopTiling2
    input C1 = 5, C2 = 10 end //Default values if C1/C2 are not defined
    output v1, v2, $jp end  // Returns parameters to other aspects

    A: select function.($l1=loop).($l2=loop){isInnermost==true}
    apply
        v1 = "tile_size_"+$l2.control_var;
        v2 = "tile_size_"+$l1.control_var;
        $l2.exec loopTiling($l2, v1);
        $l1.exec loopTiling($l2, v2);
        $jp = $l1;

        // Inserts a new variable declaration in the function scope
        // Receives the name, the type, and optionally the initialization
        $function.exec newVariable(v1, 'const int', C1);
        $function.exec newVariable(v2, 'const int', C2);
    end
end
```

Figure 3.3 DSL code for applying loop tiling over two inner-most loops using a fixed block sizes (given by the C1 and C2 inputs of the aspect).

Figure 3.4 shows a code example that could be generated from the aspect in Figure 3.3, when applied to the code in Figure 3.2. This aspect could be further refined in order to indicate more precisely which loops we want to target (e.g., name of the function containing the loops, variables of the loops). Some of the features of the aspect (e.g., loop tiling over C++, the action 'newVariable') are not implemented yet and will be developed during the ANTAREX project.

```
tile_size_i = 5;
tile_size_j = 10;
…
for (i1 = 0; i1 < N; i1 += tile_size_i) {
    for (j1 = 0; j1 < M; j1 += tile_size_j) {
        for (i = i1; i< i1 + tile_size_i && i < N; i++) {
            for (j = j1; j < j1 + tile_size_j && j < M; j++) {
                do_something(i, j);
            }
        }
    }
}
```

Figure 3.4 Example of generated code when we apply the aspect in Figure 3.3 to the code in the Figure 3.2.

The outputs of the aspect are important for modularity and composability. We want to be able to decompose aspects into smaller parts (i.e., modularity) and be able to reuse these parts as often as

possible (i.e., composability). Figure 3.5 shows an example where we use the aspect previously presented in Figure 3.3 to perform loop tiling, and then we use the outputs to feed another aspect that performs a custom tuning strategy.

```
aspectdef TileAndAutotune
    call A:LoopTiling2(5, 10); // call with aspect instance identified by A
    call myTuningStrategy(A.$jp, [A.v1, A.v2]);  // an autotuning strategy
end
```

Figure 3.5 DSL code for applying loop tiling considering tunable block sizes (with default values given by the C1 and C2 inputs of the aspect) and considering an existent autotuning strategy responsible to include the code for setup, monitor, and act.

In the following sections we describe the main knobs to be used by ANTAREX autotuning strategies.

3.1 OpenCL Tunable Parameters

With respect to OpenCL kernels, we propose eight tunable parameters based on existing research work on the autotuning of GPGPU applications. Table II summarizes the proposed parameters. The column "Control mode" identifies how the knob is controlled, by an application variable or by a compiler flag.

Table II. Summary of the OpenCL proposed knobs.

Parameter	Type	Control mode: **AV**: Application Variable **CF**: Compiler Flag	Application
device_selection	Partition of the devices set	AV	Platform selection
unroll_factor	Integer in [1, loop size]	CF	Loop transformation
loop_nest_order	Permutation on loop indices	CF	Loop transformation
tile_size	Size of the n-dimensional tile	AV	Loop transformation
cache_strategy	Partition of the variables set	CF	Local memory optimization
work_group_size	Integer in [1, CL_KERNEL_WORK_GROUP_SIZE]	AV	Workload splitting
work_items_coalesced	Integer in [1, work_group_size]	CF	Workload splitting
vector_type	Selection of vector data type	CF	Workload splitting

3.1.1 Platform Selection

A key decision when running OpenCL programs is which OpenCL device to use, when more than one is available. The CPUs can also be used as OpenCL devices, and it is possible, if supported by the application, to use multiple devices. Figure 3.6 shows an example of the OpenCL code that needs to be generated for the parameter `device_selection`. There are several ways to insert code using the DSL, for instance, by using insert actions, by using actions provided by the weaver, or with parameterizable templates.

```
Parameter: device_selection (partition of the devices set).
```

```
Example: device_selection = all CL_DEVICE_TYPE_GPU
cl_uint* device_selection; // array of device IDs
//get all platforms
cl_int err;
cl_uint numPlatforms;
cl_platform_id *platformIds;
//get number of available platform
err = clGetPlatformIDs(0, NULL, &numPlatforms);
platformIds = (cl_platform_id *) malloc(sizeof(cl_platform_id) * numPlatforms);
err = clGetPlatformIDs(numPlatforms, platformIds, NULL); //get platform IDs
for (auto platform : platformIds) {
    cl_uint numDevices; //get all devices of a given platform
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
    cl_device_id* deviceIds = (cl_device_id *)
                              malloc(sizeof(cl_device_id) * numDevices);
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, numDevices,
          deviceIds, NULL);
    device_selection = tune_partition(device_selection, deviceIDs);
}
```

Figure 3.6 Example of generated OpenCL code for parameter `device_selection`.

3.1.2 Loop Transformations

Loop transformations are known to affect directly the performance of OpenCL code. We can tune the parameters governing these transformations, in order to provide better performance and/or lower power/energy consumptions. The following are the loop transformations to be addressed with respect to OpenCL.

Loop unrolling

Loop unrolling is a classic compiler transformation, employed primarily not only to enlarge basic blocks and provide more opportunities for instruction level parallelism but also to reduce loop management overhead and improve data locality. In the context of GPGPUs, the ability to reduce

branch control instructions is critical, because these operations are more costly than in traditional CPUs [Agosta2015].

```
Parameter: unroll_factor (integer, from 1 to loop size).
```

Loop interchange

Nested loops can be mapped to OpenCL *Ndrange* dimensions, allowing their execution as a single kernel. Loop interchange is a classic compiler transformation that changes the order of loops in a nest, when possible without violating data dependencies. It can be used in OpenCL to allow better coalescing of memory accesses. Alternately, it is possible to perform transpositions of multidimensional array data, in order to achieve better data locality across work-items in the same work-group [OBoyle2013].

```
Parameter: loop_nest_order (permutation on loop indices).
```

Loop tiling

Loop tiling (see Figure 3.4) change the access pattern by creating blocks ("tiles") in the iteration space to be accessed sequentially. Loop tiling is usually exploited to improve data locality, but it can also be used to enable other optimization techniques such as inter-tile autotuning and/or parallelization [Tiwari2009].

```
Parameter: tile_size (size of the n-dimensional tile).
```

3.1.3 Cache Optimization Strategies

All the current GPGPU architectures offer a small amount of explicitly addressable cache, which is usually accessible with latencies an order of magnitude smaller than the global device memory. Their usage has been explored since the very first works in the field, and a proper exploitation of their presence is paramount to achieve a properly performing implementation. Typically, the OpenCL frameworks distributed by the producers employ the explicitly addressable caches to map the local memory of the programming model, exploiting the eventual remainder from the mapping to make up for an eventual the lack of registers required to map the work-item private variables.

Different cache strategies typically need to be developed at the application level, but it is possible to either generate local memory buffer automatically, or to select among different cache strategies implemented manually [Nugteren2015]. Figure 3.7 shows code that can be inserted by the DSL to switch between cache strategies.

```
Parameter: cache_strategy (partition of variables among the address spaces).
```

```
Example: -DTUNABLE_CACHE_STRATEGY=''__local''
#define DEFAULT_CACHE_STRATEGY "__global"
#ifndef TUNABLE_CACHE_STRATEGY
  #define TUNABLE_CACHE_STRATEGY DEFAULT_CACHE_STRATEGY
#endif
TUNABLE_CACHE_STRATEGY int var1;
TUNABLE_CACHE_STRATEGY int var2;
```

Figure 3.7 Example of generated OpenCL for parameter `cache_strategy`

3.1.4 Workload splitting

The main decision in OpenCL programs is the splitting of the workload across different processing units. We describe next three OpenCL knobs related to workload splitting.

Work-group sizing

Work-group size impacts significantly the performance of OpenCL kernels. The optimal size of the work-group is related to both the memory needs and the computation needs. In essence, the work-group size should fit the available local memory and the available cores. When there is insufficient memory, there is a tradeoff between computational resources utilization and memory efficiency [Seo2013].

```
Parameter: work_group_size (integer, in number of work-items).
```

Work-Item Coalescing

Work-group size may not be sufficient for all types of machines. In some cases, when local memory is more abundant than processing elements, sequential execution may be needed. In this case, it is preferable to coalesce work-items to fit the number of processing elements. Due to the presence of synchronization constructs, this transformation requires to restructure the work-item code so that looping over the coalesced work-items is performed within synchronization free regions [Agosta2014].

Figure 3.8 and Figure 3.9 illustrate an example of the work-item coalescing. The three parallel loops in Figure 3.8 are merged and two barriers are included. In terms of the DSL support, this transformation can be an extension of loop merge with the control where to insert the barriers. This support needs to be provided by the ANTAREX DSL and by the source to source compiler.

```
// parallel for
for (int i = 0; i < N; i++) {
    do_something_1( i );
}
// parallel for
for (int i = 0; i < N; i++) {
    do_something_2( i );
}
// parallel for
for (int i = 0; i < N; i++) {
    do_something_3( i );
}
```

Figure 3.8 Code for a sequence of loops

```
Parameter: work_items_coalesced (integer, in number of work-items).
```

```
Example: -work_items_coalesced 2
// parallel for
for (int i = 0; i < N; i++) {
    do_something_1( i );
    barrier;
    do_something_2( i );
    barrier
    do_something_3( i );
}
```

Figure 3.9 Example of generated OpenCL for parameter `work_items_coalesced` applied to the code in Figure 3.8.

Vectorization

The OpenCL language encompasses native integer and floating point vector types. The use of vector types can be seen as complementary to a proper workload split in work items and work groups. In particular, in case the target platform has an efficient thread spawning strategy, a more effective partitioning of the workload will play a key role, reducing the importance of vector types. On the other hand, in cases, such as Intel's OpenCL back end, which exploit the multicore CPUs of the host as an execution unit, we expect the use of vector types to be strongly beneficial [Agosta2015].

Vectorization options may be part of the manually generated options, or may be generated by the compiler. Figure 3.10 shows an example of OpenCL code that can be generated by the DSL to tune the vector type to be used.

```
Parameter: vector_type (selection of vector data type).
```

```
Example: -DVECTOR_T=“float4”
#define DEFAULT_VEC_T “float2”
#ifndef VECTOR_T
#define VEC_TYPE DEFAULT_VECTOR_T
#endif
VECTOR_T my_vector_variable;
```

Figure 3.10 Example of generated OpenCL for parameter `vector_type`

3.2 C/C++ Tunable Parameters

The C/C++ code has associated a number of knobs. Table III presents the parameters that are controlled (statically or dynamically) at the C/C++ source code level (i.e., they are exposed and guide the C/C++ source to source compiler). As we believe the transformations and tunable parameters associated are more common than the ones regarding the OpenCL code (previous section) we do not include in this section a description of each of the parameters presented in the table.

Table III. Summary of the C/C++ proposed knobs.

Parameter	Description	Control mode: **SC**: Statically Controlled **DC**: Dynamically Controlled	Control mode: **AV**: Application Variable **CF**: Compiler Flag	Type
Perforation_factor	The number of iterations to skip, or the number of iterations to perform before skipping	DC	AV	Loop transformation
unroll_factor	Number of iterations to unroll	DC/SC	CF	Loop transformation
loop_nest_order	Permutation on loop indices	DC/SC	CF	Loop transformation
tile_size	Size of the n-dimensional tile	DC/SC	AV	Loop transformation
cache_strategy	Partition of the variables set	DC	AV	Local memory optimization
splitting_location	Split a loop in two or more loops	SC	AV	Loop transformation
merge_ids	Merge two or more loops	SC	AV	Loop transformation
versioning_ids	Provide two or more versions of a loop, being the one to be executed at	DC	AV	Loop transformation

Parameter	Description	Control mode: **SC**: Statically Controlled **DC**: Dynamically Controlled	Control mode: **AV**: Application Variable **CF**: Compiler Flag	Type
	a particular execution point selected at runtime. Number of versions and strategy to provide each version.			
peeling_iteration_space	Extract some iterations of the loop to a prologue or an epilogue.	SC	AV	Loop transformation
unrolljam_factor	Unroll an outer loop by a factor K and jam (merge) the K inner loops resultant from the unrolling. Integer in [1, loop size[	SC	AV	Loop transformation
Inline	Inline a function. Function id	SC	AV/CF	Function transformation
Multiversioning_ids	Provide different implementations of a function and select at each call the one to be executed at runtime. Number of versions and strategy to provide each version.	DC	AV	Function transformation
Memoization_scheme	Size of the table. Enabling/disabling memorization based on runtime information.	DC	AV	Function transformation
approximation_scheme	Kind of approximation method.	DC	AV	Function transformation
Outlining_locs	Section of code to move to a function.	SC	AV	Function transformation
Partitioning_ids	Functions to be moved to accelerators. Associated to strategies that control the interface and the data communication.	SC	AV	Partitioning transformation
var_id	Dynamic range [min, max] wordlength (WL), fraction bits float128, float, double, int, long	SC	AV	Type conversions

Parameter	Description	Control mode: **SC**: Statically Controlled **DC**: Dynamically Controlled	Control mode: **AV**: Application Variable **CF**: Compiler Flag	Type
Application-specific knobs	Application input parameters (e.g., number of trials into a Monte Carlo simulation, threshold values)	SC/DC	AV	Refactoring

3.3 Observation Strategies

One important feature of the ANTAREX approach is to provide support for developers to define strategies that specify where to monitor in the input application, the data to be collected and reported (and how), and the periodicity of the monitoring/measurements. LARA libraries will manage the use of the monitoring library (provided in WP3) and the properties related to the measurements to be done (e.g., execution time, energy, power, temperature), their periodicity, and the possible policies to aggregate data from different nodes and the nodes to be involved.

3.4 Summary

The ANTAREX project intends to provide a holistic approach for runtime adaptivity and autotuning. The analysis presented in this section provides the main features related to the adaptivity and autotuning requirements and to require extensions to the pre-ANTAREX LARA status.

4 The LARA Cross-Layer Programming Language for Strategies

LARA [Cardoso2014] is a domain-specific aspect-oriented programming language (AOP) [Kiczales1997][Fabry2015] specially designed for allowing developers to program code instrumentation strategies, to control the application of code transformations and compiler optimizations, and to effectively control different tools in a toolchain. LARA provides a separation of concerns, including non-functional requirements and strategies, for the mapping of high-level applications to high performance heterogeneous embedded systems. Moreover, LARA provides developers with artefacts and with a unified view, which diminishes the efforts to program and evaluate design space exploration (DSE) schemes. LARA brings a novel approach to AOP that is partially agnostic to the target language. LARA aspects allow the description of sophisticated code instrumentation schemes (e.g., for profiling customization) [Cardoso2014], advanced selection of critical code sections for hardware acceleration [Cardoso2013a], advanced mapping strategies including conditional decisions based on hardware/software resources [Cardoso2013b], of sophisticated sequences of compiler transformations [Cardoso2012b], of strategies determining sequences of compiler optimizations [Martins2014], and of strategies targeting multicore architectures. Furthermore, LARA provides mechanisms for controlling all elements of a toolchain in a consistent and systematic way, using a unified programming interface [Cardoso2013b]. LARA allows developers to capture nonfunctional requirements from applications in a structured way, leveraging high/low-level actions and flexible toolchain interfaces. Developers can thus benefit from retaining the original application source-code while exploiting the benefits of an automatic approach for various domain-specific and target component-specific compilation/synthesis tools. Specifically, the LARA AOP approach has been designed to help developers reach efficient implementations with low programming effort.

The experiences of using aspects for software/hardware transformations [Cardoso2013a] have revealed the benefits of AOP in application code development, including: program portability across architectures and tools, and productivity improvement for developers. Moreover, with the increasing costs of software development and maintenance as well as verification and validation, both these two facets of application code development will continue to be of paramount significance to the overall software development cycles. In essence, LARA uses AOP mechanisms to offer in a unified framework:

(a) a vehicle for conveying application-specific requirements that cannot otherwise be specified in the original programming language,

(b) an approach for using the above mentioned requirements to guide the application of transformations and mapping choices, thus facilitating design-space-exploration (DSE), and

(c) interfacing in an extensible fashion the various compilation/synthesis components of the toolchain.

LARA has been extended by supporting tools including a LARA interpreter and weavers. The LARA technology has been used in four compiler tools: MATISSE (a MATLAB to C compiler being developed at Univ. of Porto/INESC-TEC), Harmonic (a source-to-source compiler being developed at Imperial College London, UK), MANET (a source-to-source compiler being developed at Univ. of

Porto/INESC-TEC) and ReflectC (a CoSy-based C compiler developed at Univ. of Porto). The research and development of the LARA language in the context of the FP7/REFLECT research project [Cardoso2013a] has been strongly driven by industry requirements (e.g., Honeywell, Coreworks, and ACE). Previous LARA-based tools have been evaluated with real-life industrial application C codes and the experimental results provided strong evidence of its usefulness and practical benefits in terms of application and performance portability as well as programmer productivity and portability across distinct target architectures. We see the flexibility of DSLs, such as LARA, as a key programming technology that will enable developers to meet increasingly demanding challenges in developing future high-performance systems.

4.1 Overview

The LARA DSL [Cardoso2012a] [Cardoso2014] has been designed to be a general purpose aspect-language, as agnostic to the target language as possible and it has the capability to address concerns such as code instrumentation [Pinto2015], compiler optimizations [Martins2014], mapping decisions [Cardoso2013a] and type and code specialization [Bispo2013]. Although its main application has been to transform C programs [Pinto2015], it has been used to generate C and OpenCL implementation from higher-level languages, such as MATLAB [Bispo2013]. Using LARA, developers specify source code transformations (e.g., loop unrolling, loop tiling, unroll-and-jam, code insertion) and annotations using aspects. A compiler enhanced with a LARA weaver can interpret these aspects and carry out the corresponding transformations. This process generates a modified source code that is then compiled using the native target architecture compilation system. These aspects are fully parameterisable and composable, allowing developers to build a wide range of transformation "recipes" without having to explicitly manipulate the original source code.

4.2 Current Status and Approach

Table IV presents the ANTAREX most relevant source to source compilers with LARA support. In the context of the C source to source compiler controlled by LARA, the Harmonic version extended with LARA [Cardoso2013b] has achieved an advanced status as a result of the development efforts provided during the FP7 REFLECT project. However, part of the compiler was developed by members of the Imperial College in London and it is not an open-source compiler. Although the use of the compiler in the ANTAREX project might be possible, an analysis in the beginning of the project has shown us that the requirements in ANTAREX would need significant extensions and maintenance costs and the ROSE framework[1] used in the compiler and its support and dependences of specific libraries were factors that contributed to a decision to focus on a new source to source compiler.

MANET is another C source to source compiler, guided by LARA, being used. This compiler is based on the Cetus framework[2] with the major disadvantage to only deal with ANSI C and extensions to manipulate recent C versions and C++ code would require many work efforts. However, we have

[1] ROSE compiler infrastructure, Lawrence Livermore National Laboratory (LLNL), USA. http://rosecompiler.org/

[2]CETUS: A Source-to-Source Compiler Infrastructure for C Programs, Purdue University, USA. https://engineering.purdue.edu/Cetus/

decided to continue extending and using MANET in the context of ANTAREX as a way to apply and evaluate some ideas.

The language itself has achieved a high maturity level result of the work being done over the years and of the decision to adopt the overall syntax and semantic of JavaScript for the imperative parts of the LARA aspects.

Table IV. ANTAREX relevant source to source tools with LARA support.

Contributions	Associated Projects	Relevance to ANTAREX
C to C source to source compiler: MANET	The MANET compiler has been developed with partial financial support from AutoSeer [Autoseer2015], RL8 [RL8-15], and a PhD project.	Selected as a tool to evaluate some of the LARA strategies and new concepts.
C to C source to source compiler: HARMONIC+	REFLECT FP7 Project [REFLECT10] [Cardoso2013]	Not considered as an ANTAREX tool. Decision was previously made based on the analysis of the existent source to source tools, frameworks used, regularity of updates and conformance to C++; dependability on specific libraries and frameworks, etc.

4.3 LARA and the Related Work

Research efforts (e.g., ChiLL [Hall2009], POET [Yi2012a], the search-space description language – SSDL – in [Kim2015]) exposed source-level transformations in a controlled fashion by offering a script specification of the sequence of transformations and corresponding parameters. These frameworks differ from the LARA-based approach in various respects. First, LARA's domain of applicability extends beyond scientific and engineering codes. Second, the LARA-based approach bridges the gap between hardware and software compilation. Lastly, by directing transformation engines, LARA allows programmers to customize, in a non-intrusive way, the source code to be used by other tools.

4.4 Summary

The focus of previous LARA work (i.e., pre-ANTAREX) was high-performance embedded computing systems consisting of a CPU and multiple hardware cores implemented in FPGAs. The strategies mainly addressed suitable code transformations for mapping computations to hardware accelerators using FPGAs, phase selection and phase ordering in the context of a single CPU core, and instrumenting/monitoring strategies.

However, the analysis of the ANTAREX use cases requirements (in WP2) revealed that the LARA language is a DSL that is mostly capable of fulfilling those requirements, especially in terms of its approach, syntax and semantics. We identified the extensions needed to fully comply with what is expected in the ANTAREX project. Those extensions include the support for dynamic apply sections, identified by the keyword, *dynamic*, the support for C++, new code transformations and analysis, and functions that detect new attributes (such as the *parfor* and *reduce* for parallelization strategies).

5 Programming Model and DSL Features to Address the ANTAREX Requirements

The intention of the ANTAREX project is to provide a programming model to specify runtime adaptivity strategies and code transformations and optimizations to be applied to an input software application. Another intention is to provide support to developers in different phases of the code optimization process. All these intentions are aligned with non-functional requirements as they express complementary concerns over the application logic. In the ANTAREX approach the application source code already exists (or is being developed but without mixing adaptivity behavior with the main functionalities) and users need to optimize the application in order to provide higher performance and/or low energy/power consumption. The optimizations might deal with code transformations/optimizations, compiler optimizations, and with runtime autotuning/adaptivity.

5.1 Specification of the Programming Model and DSL

The ANTAREX programming model is focused on a DSL devoted to express those needs and to interplay with tools/compilers in order to automatically apply those concerns via the use of DSL strategies (considered as "recipes"). Being the intention to provide support to secondary concerns and "recipes" (in most cases reusable by many applications) it is intuitive that a programming model based on a separation of concerns would suit the ANTAREX needs.

The DSL needs to provide mechanisms to identify code constructs and application execution points and to specify actions on those application points. The tool flow would need to support the application of the DSL "recipes" to the application source code (e.g., via a source to source compiler). In addition, the DSL needs to support the initialization of some of the components used in the system such as the autotuner or during the exploration of different alternatives.

- Enable **separation of concerns:** non-functional concerns (performance, energy, monitoring) are decoupled from the application code (functional description);
- Useful to express **strategies** for instrumentation and synthesis/compiler optimizations;
- Fully explore **compiler optimization sequences** according to code and target architectures;
- Support **Design Space Exploration** mechanisms to fully explore compiler optimizations;
- Enable more advanced **control** than using pragmas/ directives/switches.

5.2 LARA Extensions Regarding the ANTAREX DSL Requirements

The combination of declarative and imperative models in the same language as provided by LARA seems to be suitable as the selection of program execution points and code locations are naturally represented as a declarative query-based style and the steps related to the actions to be applied are more natural to be expressed in the imperative style.

The LARA join point model previously used was dedicated to imperative programming languages without support to classes and objects. ANTAREX needs the support to deal with the OO C++ language and thus the join point model is being extended by join points to express uses of objects, classes, and methods.

One of the most important features of LARA are the functions associated to joinpoints which allow them to return information about code attributes. The language already has flexible way to add additional functions as long as the underlying compiler can provide information about the attributes to return. In ANTAREX, there is the need to support attributes that enable the specification of LARA strategies regarding the OpenMP directive-driven programming model, MPI, and the partitioning of data and computations.

The use of LARA in the context of runtime autotuning was very preliminary. This needs the support of code transformations specially extended to expose tuning parameters in a form of software knobs and the support to prepare the application code to integrate the autotuner.

Some of the support foreseen will be through LARA APIs and will provide the abstraction levels required to help developers. An API example is the one mentioned in the context of the Margo autotuner (next Section).

5.3 LARA Support for Autotuning

The programming model envisioned for ANTAREX need to:

- Define mechanisms and strategies to be adopted at application-level to on-line adapt the application behavior and the platform configuration with respect to changing workloads, operating conditions and computing resources;
- Search for the best combination of the software knobs (i.e., application parameters, code transformations and code variants) impacting the performance and energy efficiency of the application;
- Provide DSL support to instrument the application for monitoring and to support autotuning.

LARA strategies will be specified in order to monitor an application and to make the measurements related to the operating points to be used by the autotuner. The operating points and the measurements must be output to an XML file processed by Margo (WP3). This provides the first stage needed by the autotuner in a flexible way regarding the measurements as the LARA aspects will provide the possibility to specify the strategy and the mechanisms for measuring the operating points.

Table V shows for each of the main Margo artefacts the API actions possible to control at the application source code. LARA aspects will provide the woven of the application source code with the required calls to the Margo API. Figure 5.1 shows an example of how LARA will look like for initializing the Margo autotuner. In this example we do not present a LARA aspect with parameters and the aspect is specific for a particular example named as “EX”. Most Margo parameters can be encapsulated in other LARA aspects or LARA objects in order to make simpler the code seen by most developers. Figure 5.2 presents a different version considering the existence of a LARA aspect defined as MargoSetup.

Through LARA the user will have an option to easily evaluate different autotuning policies, parameters, and the periodicity of the tuning and of the observability. All options can be encapsulated in LARA aspects and in ANTAREX we will provide a LARA API for Margo in order to make even easier its use by developers.

Table V. Margo autotuner parameters and actions controlled from the application using the API.

Margo autotuner main artefacts	**API Actions**
State	Create/Select/Delete a state
Constraint	Add/Remove a constraint
Rank	Define the rank function
Goal	Create (destroy) a goal
	Change the value of a goal, thus on all the related constraints
	Specify during its creation a goal as dynamic or static
	The name of either the target software-knob, using the attribute knob_name, or the target metric, using the attribute metric_name. Both names must refer to a field of the Operating Point.
	The target statistical property of the monitor, using the attribute dFun. The available statistical properties are the ones computed by the monitor
	The actual target value of the goal
	Target comparison function of the goal, using the attribute cFun. The available comparison functions are “GT” (greater than), “GE” (greater or equal than), “LT” (less than), “LE” (less or equal than).
Operating Points	Add/Remove Operating Points
Software Knobs	Define the name of the target software-knob, using the attribute knob_name. The name must refer to a field of the Operating Point.
Optimization	Policy of the application as a multi-objective constrained problem. Specify the attribute combination, which define how to combine together the components of the rank. In the current implementation the Autotuner supports the following combination: linear or geometric.

```
Aspectef MargoSetup4EX
     initialize
          var Knob =    {name:”num_threads”,
                        var_name: “threads”, var_type:”int”};

          var Goal =    {name:”EX_responseTime_goal”,
                        monitor:“EX_responseTime_monitor”,
                        dFun:”Average”, cFun:”LT”, value:”EX_SLA”};

          var State =   {name:”power_optimized”, starting: “yes”,
                        minimize:{    combination:”linear”,
                                      metric_name:”EX_avg_power”,
                                      coef:”1.0”},
                        subject:{to:“EX_responseTime”,
                                 metric_name:”EX_responseTime”,
                                 priority:”1”}
                        }
          Margo.setup(Knob, Goal, State, “autotuning_spec.xml”);
     end
end
```

Figure 5.1. LARA code for an example of initialization of the Margo autotuner.

```
Aspectef MargoSetup4EX
     initialize
          var Setup = new MargoSetup();
          Setup.Knob.name = “num_threads”;
          …
          Setup.Goal.dFun = “Average”;

          call Setup();
     end
end
```

Figure 5.2. Another version of the LARA code for an example of initialization of the Margo autotuner considering the existence of the MargoSetup aspect.

5.4 LARA Support for Approximate Computing

We will use the generic source-to-source capabilities of the ANTAREX tool flow and DSL to develop strategies that add support for approximate computing. For instance, in collaboration with the domain experts from #UC1, we have already found opportunities for approximate computing that could provide performance increase and energy savings (e.g., switching between distance functions; skipping loop iterations – loop perforation). This would require LARA strategies: (a) to replace functions with implementations with lower accuracy (considering a different algorithm, an

approximate function, memorization schemes using ranges of values for accessing tables instead of single values, the use of data types with lower precision) and the possibility the strategy generates code for the new approximate computing function; (b) to expose and control some software knobs, such as the number of iterations of an algorithm and the loop perforation factor.

The capabilities of the ANTAREX tool flow and approach to deal with approximate computing is based on the AOP approach provided by LARA to modify code as a result of a weaving process with the application code and the aspects as inputs and on the LARA strategies to deal with approximate calculations and to be specially developed for the ANTAREX project.

5.5 LARA Extensions

In this section we briefly describe the planned work regarding the DSL and the tool support. We start by summarizing the background regarding the DSL, the techniques, and the tools being used in the ANTAREX project.

One of the main strengths of the ANTAREX project is the combination of teams that bring to the consortium know-how and previous work suitable to help on fulfilling the main ANTAREX objectives. The existence of techniques and/or tools previously developed and possibly to be used with minor adaptations and/or extensive extensions is an important aspect for the success of the project. Table VI summarizes the main contributions of the work being done in WP2 and their status before the beginning of the ANTAREX project.

Table VI. Identification of the major contributions of WP2 and their status before ANTAREX.

Contributions	Project which contributed most	Extensions/improvements in the context of ANTAREX
C/C++ Source to Source Compiler: not existent	Compiler not existent and being developed in ANTAREX	ALL (including the LARA support for C++ object-oriented constructs)
C/C++ and OpenCL Translation and Integration	ANTAREX	ALL (including the LARA support for OpenCL strategies and the compiler support to modify OpenCL code and to translate a subset of C/C++ code to OpenCL)
Type conversions (e.g., floating to fixed-point)	Possible use of the ID.Fix[3] tool for some experiments regarding the floating to fixed-point conversions.	The use of fixed-point is under evaluation in terms of their impact and need in the context of the CPUs and GPUs used.
Apply LARA sections fully migrated to runtime (via dynamic)	Work proposing a dynamic version for LARA started in [Carvalho2015] in the context of a PhD project.	ALL (including the migration of a LARA subset to target code)

[3] http://idfix.gforge.inria.fr/doku/doku.php?id=home

LARA specific support to OpenMP strategies	ANTAREX	ALL (including compiler analysis to support specific attributes such as parfor, reduce, private)
LARA specific support to MPI strategies	ANTAREX	ALL
LARA specific support to OpenCL strategies	ANTAREX	ALL
LARA support for multiversioning	ANTAREX	ALL
LARA specific support to memoization strategies	Start with the work on memoization of "pure" functions by INRIA [Suresh2015]. New approach using LARA and memorization techniques is being R&D in ANTAREX	New memorization techniques; Support provided by LARA (for monitoring and generation of the functions, and for preparation of the application code to use the memoized functions)
LARA specific support to approximate computing strategies	ANTAREX	ALL (this may require LARA strategies to replace functions with implementations with lower accuracy and the possibility the strategy generates code for the new approximate computing function).
LARA specific support to autotuning strategies (including code transformations to expose knobs, interface and communication to autotuner)	Work started with the R&D previously done in the context of the Argo framework[4] [Gadioli2015]. We anticipate major contribution from ANTAREX.	Extensions and improvements of the Argo framework and the use of LARA strategies to guide the autotuning.
Framework for runtime exploration of compiler flags	ANTAREX	ALL
Selection of compiler flags	Work started with the R&D previously done by POLIMI [Ashouri2014]	ANTAREX contributions already in [Ashouri2016]. To be extended to other compilers such as icc; in the context of energy efficiency; with other approaches;
Phase selection and phase ordering	UPORTO work started with the RL8 project and with two PhD projects. [Martins2014] [Martins2016][Nobre2016]	Reducing the number of evaluations/explorations using execution; Split-compilation approach; in the context of energy efficiency; with other approaches;
Monitoring	ANTAREX	LARA libraries for dealing with the monitoring capabilities of the API proposed by WP3.

[4] Work was supported in part by the EC under the grants HARPA FP7- 612069 and CONTREX FP7-611146.

5.6 LARA Language

The LARA language is being extended with the possibility to specify *dynamic* applies. The language has been updated in terms of some reserved words such as *exec* and *replace*. The attribute model needs to be extended with features related to the support of OpenMP strategies, such as the inclusion of the *parfor* and *reduce* attributes. We expect that during the ANTAREX project other attributes might be needed.

5.6.1 LARA Frontend

The LARA front end is the tool that processes the input LARA code and according to the join point, attribute, and action models generates Aspect-IR descriptions.

We have started a number of improvements in the LARA frontend. Actions are now able to return values, in contrast to the previous versions where all actions returned void. This extension is very important for some code transformations such as “function cloning”. This action can now return the new name given to the cloned function or return a join point reference to the newly added function. We have added a utility class named "LaraObject" to the LARA language and interpreter. This class provides useful containers to be used with LARA aspects. The idea of this object is to use tuples captured with selects, such as <caller, callee> or <function, var>, and use them as identifiers to store information. Due to the importance of the “for...of” type of loop (defined in the ECMAScript 2015 general purpose programming language standard) for iterating over the elements of collections, we decided to added it to the LARA language, and is now supported by larai (the LARA interpreter). Moreover, LARA technology includes now an editor that is automatically linked to the weaving engine. Complementary to the command line execution, larai has now an integrated graphic user-interface (GUI) in which the LARA strategies can be developed and the weaving options are selected. LARA extensions have been identified according to WP1 activities.

An extension to the LARA frontend is related to the analysis of the code when the LARA aspects include dynamic applies. The analysis needs to take into account to the LARA subset considered for translation (it will be based on the LARA strategies developed in the second half of the project) and to the identification of possible conflicts regarding the use of dynamic code sections vs static code sections.

5.6.2 LARA Interpreter

The LARA interpreter (larai) is the engine responsible to execute LARA strategies and to communicate with weavers. We have finished updating the LARA interpreter to execute in the Nashorn JavaScript engine[5]. This update provides a faster engine and fixes some previous issues. We have also updated the exception messages, providing a stack trace-based message including information about the invoked aspect, the executing apply, and the action that is being performed. The new version includes functionalities that provide information regarding the target language specification. It is now possible to verify the selectable join points, attributes and actions of a given join point. A new concept of “gears” was added to larai. These gears are similar to event listeners that

[5] https://en.wikipedia.org/wiki/Nashorn_(JavaScript_engine)

respond to LARA-related events. For this, several events were added to LARA, including: starting and finishing the weaver execution, entering or exiting a call/select/apply/cmd, join point production, among others. Gears can be used to add secondary features to the weaving process, such as debugging, weaving statistics (e.g., number of selected and/or advised join points), or to delay actions (e.g., store and apply actions only after the aspect is exited).

Current work is being focused on adjusting the run/cmd statements/definition to include a more sophisticated approach to control the execution of tools. Specifically, we are adding the utility of output redirection in order to define if the output should be directly to the console, to a given target file, or just ignored. Furthermore, a JSON (JavaScript Object Notation) report will be generated consisting of the following information: return status, report from the tool executed, console output, and exception (if any). The execution of tools is an important LARA feature in the context of the split-compilation, and design-space exploration (DSE).

5.6.3 LARA to Target Translation

In the presence of dynamic applies we need a tool to translate a subset of LARA to the target code (C++ in the ANTAREX case). Some LARA code in aspects including dynamic applies and the code in those dynamic apply sections will be translated to C++ code to be statically compiled and linked to the application code. We intend to evaluate the use of embedded code vs the use of threads responsible for the execution of the strategies and communicating with the application code. During the project we also have the intention to evaluate the use of the LARA interpreter in runtime vs the use of the target code generated from LARA.

5.6.4 C/C++ Source to Source Compiler

One important stage of the ANTAREX toolflow is the source to source compiler. The requirements elicited in WP1 pointed to a C/C++ source to source compiler and a previous analysis showed the inexistence of a suitable C/C++ compiler for the ANTAREX needs. This conducted to the development of a C/C++ source to source compiler with LARA support and including clang[6] as front-end. This compiler will be responsible to analyze the application code and weave the C/C++ application code according to the LARA strategies.

Regarding the C++ weaver, we have started the analysis of the clang front-end in terms of its software architecture, and features. Clang has been selected as a result of an analysis of most relevant existent source to source compilers with C/C++ as input language and will be one of the source to source tools in the ANTAREX tool flow.

We started to analyze how to integrate Clang into a C++ Clang-based weaver. We considered two prototypes, one based on socket communication between a Java weaver and a C++ tool based on Clang, and another based on instructing Clang to dump information and then parse it in Java. We concluded the latter approach could be more advantageous and we are currently developing that solution. Although the work is very preliminary, currently we are able to parse more than 40 C++ AST nodes and generate C++ code for some simple programs.

---

[6] http://clang.llvm.org/

Using the previous C weaver, we were able to develop LARA aspects to add concerns for timing, OpenMP-based parallelism, runtime specialization and autotuning using the ANTAREX framework being developed (i.e., Versioning Compiler and Margo libraries). The support of floating to fixed-point conversions, through the use of libraries such as libfixmath[7] or the use of specific compilers to automate the floating to fixed-point conversion process such as ID.Fix[8] from INRIA will be analyzed.

One important feature of the source to source compiler is the possibility to deal with OpenCL code. The ANTAREX approach will consider the LARA support for OpenCL strategies and the compiler support to modify OpenCL code and to translate a subset of C/C++ code to OpenCL. The OpenCL generation will be based on C++ templates and on the input C/C++ code with the OpenCL constraints. This ANTAREX feature will take as example and inspiration the OpenCL code to be provided for some of the hotspots of the application use cases. We would like to note that the intention is not to provide an advanced C/C++ to OpenCL code translation system but to provide the required support to integrate OpenCL code and the translation of some C/C++ code to OpenCL in order to evaluate the ANTAREX adaptivity dimensions for the application use cases.

5.6.5 Split Compilation Framework

The libVersioningCompiler library provided by POLIMI (in the context of WP2) serves as the starting point for the development of the ANTAREX compiler infrastructure. The main goal of this version is to provide the ability to generate and execute specialized versions of the code. Specialization can take the form of custom code, precision tuning for floating point computations, or enabling OpenMP with different number of threads. A first version of the library is developed in pure C, to support existing versions of the benchmark and LARA language. The second version of the libVersioningCompiler library provides a more refined C++ interface, as well as the ability to support split compilation and phase ordering in addition to specialization.

5.6.6 Library of LARA Strategies

One of the important contribution of the ANTAREX project will be provided by a library of strategies specified in LARA and possible to be applied by developers according to their needs. Through the duration of the project LARA strategies will be added to the library and will be available to all the partners. In the middle and in the end of the project the library will be publically available through the ANTAREX website.

The library of LARA strategies will provide LARA aspects for also dealing with the Margo autotuner (WP3) and with the monitoring system (WP3).

5.7 Summary

This section described the programming model to be used in the ANTAREX project and how that programming model will be supported both in terms of the DSL and in terms of the supporting tools. The programming model is inspired on AOP and in the separation of concerns. The programming

[7] https://code.google.com/archive/p/libfixmath/

[8] http://idfix.gforge.inria.fr/doku/doku.php?id=home

model focused on by ANTAREX is related to the specification of runtime adaptivity strategies, code optimizations, and strategies to monitor applications and system metrics (e.g., power, energy, and temperature). The analysis conducted strongly shown us the suitability of LARA as the DSL to support the programming model. However, the language and the supporting tools need to be extended in order to make some aspects of the programming model a reality and in order to provide additional support to developers.

6 Conclusions

This report presented the ANTAREX approach in terms of programming model and DSL (Domain-Specific Language). The report is a result of a deep analysis of representative approaches for autotuning and for runtime adaptivity over the software stack, from the application code to the runtime environment. The analysis included the ANTAREX use case requirements and other more generic requirements from the two Supercomputing Centers (CINECA and IT4I) involved in the project. The main idea is to provide a programming model useful for a wide range of applications and development/optimization stages, with a fast learning curve, easy to use and to apply, and with the flexibility needed to be extended and to be supported by other tools and custom libraries.

The extensions and/or features proposed for the LARA DSL and for the supportive tools are focused on those criteria and tried to be minimal in terms of the LARA grammar.

7 References

[Agosta2014] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2014. Towards Transparently Tackling Functionality and Performance Issues across Different OpenCL Platforms. In Proceedings of the 2014 Second International Symposium on Computing and Networking (CANDAR '14). IEEE Computer Society, Washington, DC, USA, 130-136. DOI=http://dx.doi.org/10.1109/CANDAR.2014.53

[Agosta2015] Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico, and Gerardo Pelosi, 2015. OpenCL performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives. Concurrency Computat.: Pract. Exper., 27: 3633–3660. doi: 10.1002/cpe.3358.

[Agosta2014] G. Agosta, A. Barenghi, G. Pelosi, M. Scandale. Towards Transparently Tackling Functionality and Performance Issues Across Different OpenCL Platforms. In proceedings of the Second International Symposium on Computing and Networking — Across Practical Development and Theoretical Research (CANDAR 2014), December 2014

[Altera2013] Altera. Altera SDK for OpenCL - Getting Started Guide. Altera, San Jose, CA, USA, November 2013. http://www.altera.com/literature/hb/ opencl-sdk/aocl_getting_started.pdf.

[Ansel2009] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming* Language Design and Implementation (PLDI '09). ACM, New York, NY, USA, 38-49. DOI=http://dx.doi.org/10.1145/1542476.1542481

[Ashouri2014] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Cristina Silvano: A Bayesian network approach for compiler auto-tuning for embedded processors. ESTImedia 2014: 90-97

[Ashouri2016] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, Cristina Silvano: COBAYN: Compiler Autotuning Framework Using Bayesian Networks. TACO 13(2): 21 (2016)

[Autoseer2015] AutoSeer: Automated Test Oracles for Software Error Detection (PTDC/EIA-CCO/116796/2010), 2012 - 2015.

[Beyer2011] James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski. OpenMP for accelerators. IWOMP 2011, LNCS 6665, pp. 108–121, 2011.

[Bigus2002] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills, W. N., and Diao, Y.. Able: a toolkit for building multiagent autonomic systems. IBM Systems Journal, 41(3):350–371.

[Bispo2013] João Bispo, Pedro Pinto, Ricardo Nobre, Tiago Carvalho, João M. P. Cardoso, Pedro C. Diniz, "The MATISSE MATLAB Compiler - A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms," in IEEE International Conference on Industrial Informatics (INDIN'2013), Bochum, Germany, 29-31 July 2013.

[Cardoso2013] J.M.P. Cardoso, P. Diniz, J.G. Coutinho, Z. Petrov (eds.), Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach, Springer, May 2013.

[Cardoso2012a] João M. P. Cardoso, Tiago Carvalho, José Gabriel de F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro C. Diniz, Zlatko Petrov, "LARA: An Aspect-Oriented Programming Language for Embedded Systems," in International Conference on Aspect-Oriented Software Development (AOSD'12), Potsdam, Germany, March 25-30, 2012, ACM, pp. 179-190.

[Cardoso2012b] J.M.P. Cardoso, et al., "Specifying Compiler Strategies for FPGA-based Systems," in Proc. 20th Annual IEEE Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM'12), Toronto, Ontario, Canada, April 29-May 1, 2012, pp. 192-199.

[Cardoso2013a] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Ricardo Nobre, Razvan Nane, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Koen Bertels "Controlling a Complete Hardware

Synthesis Toolchain with LARA Aspects," In Elsevier Journal on Microprocessors and Microsystems, Volume 37, Issue 8, Part C, Nov. 2013, pp. 1073-1089.

[Cardoso2013b] J.M.P. Cardoso, P. Diniz, J. G. Coutinho, Z. Petrov (eds.), Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach, Springer, 1st edition, May 2013.

[Cardoso2014] João M. P. Cardoso, José G. F. Coutinho, Tiago Carvalho, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves, "Performance Driven Instrumentation and Mapping Strategies Using the LARA Aspect Oriented Programming Approach," in Software: Practice and Experience (SPE), John Wiley & Sons Ltd ("Wiley"), USA, Dec. 2014.

[Carvalho2015] Tiago Carvalho, Pedro Pinto, João M. P. Cardoso: Programming Strategies for Contextual Runtime Specialization. SCOPES 2015: 3-11.

[Casanova2011] Casanova, P., Schmerl, B. R., Garlan, D., and Abreu, R.. Architecture-based run-time fault diagnosis. In Proceedings of the 5th European Conference Software Architecture, ECSA'11, pages 261–277.

[Casanova2013] P. Casanova, D. Garlan, B.R. Schmerl, R. Abreu: Diagnosing architectural run-time failures. SEAMS 2013, pp. 103-112.

[Charfi2009] Charfi, Anis, Tom Dinkelaker, and Mira Mezini. "A plug-in architecture for self-adaptive web service compositions." Web Services, 2009. ICWS 2009. IEEE International Conference on. IEEE, 2009.

[Cheng2012] Cheng, Shang-Wen, and David Garlan. "Stitch: A language for architecture-based self-adaptation." Journal of Systems and Software 85.12 (2012): 2860-2875.

[Cunha2006] Cunha, C., Sobral, J., Monteiro, M., Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, In Proceedings of the 5th international conference on Aspect- oriented software development (AOSD '06), Bonn, Germany, March 2006.

[Dagum1998] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Comput. Sci. Eng. 5, 1 (January 1998), 46-55. DOI=10.1109/99.660313 http://dx.doi.org/10.1109/99.660313

[DeKleer1987] De Kleer, J. and Williams, B. C.. Diagnosing multiple faults. Artificial Intelligence, 32(1):97–130.

[Delaval10] G. Delaval, H. Marchand, and E. Rutten, "Contracts for Modular Discrete Controller Synthesis," In Proc. ACM Conf. on Languages, Compilers, and Tools for Embedded Systems, LCTES'10, pp. 57-66, 2010.

[Ding2015] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15). ACM, New York, NY, USA, 379-390. DOI: http://dx.doi.org/10.1145/2737924.2737969

[Diniz1997] P.C. Diniz, and M.C. Rinard, "Dynamic Feedback: An Effective Technique for Adaptive Computing," in SIGPLAN Not. 32, 5 (May 1997), pp. 71-84.

[Du2012] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Comput. 38, 8 (August 2012), 391-407.

[Elangovan2012] V. Elangovan, Badia, R. M., and Ayguadé, E., "OmpSs-OpenCL Programming Model for Heterogeneous Systems", 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC2012). Tokyo, Japan, 2012.

[Elangovan2015] VK Elangovan, RM Badia, E Ayguadé, Auto-Tuning OmpSs-OpenCL Kernels Across GPU Machines,Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA-DITAM '15), Pages 31-36.

[Fabry2015] J. Fabry, T. Dinkelaker, J. Noyé, and É. Tanter, "A Taxonomy of Domain-Specific Aspect Languages," *ACM Comput. Surv.*, vol. 47, no. 3, p. 40:1–40:44, Feb. 2015.

[Gadioli2015] Davide Gadioli, Gianluca Palermo, Cristina Silvano: Application autotuning to support runtime adaptivity in multicore architectures. SAMOS 2015: 173-180

[Garlan2001] Garlan, D., Schmerl, B., and Chang, J.. Using gauges for architecture-based monitoring and adaptation. In Proceedings of the 2001 Working Conference on Complex and Dynamic Systems Architecture.

[Garlan2004] D. Garlan, et al. Rainbow: Architecture-based self adaptation with reusable infrastructure. IEEE Computer, 37(10), October 2004.

[Ghosh2007] Ghosh, D., Sharman, R., Rao, H. R., and Upadhyaya, S.. Self-healing systems - survey and synthesis. Decision Support Systems in Emerging Economies, 42(4):2164–2185.

[Gray2012] Grauer-Gray, Scott, et al. Auto-tuning a high-level language targeted to GPU codes. Innovative Parallel Computing (InPar), pp. 1-10, 2012.

[Hall2009] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2009. Loop transformation recipes for code generation and auto-tuning. In Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing (LCPC'09), Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer-Verlag, Berlin, Heidelberg, 50-64.

[Han2009] Tianyi David Han and Tarek S. Abdelrahman. 2009. hiCUDA: a high-level directive-based language for GPU programming. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2). ACM, New York, NY, USA, 52-61. DOI=10.1145/1513895.1513902 http://doi.acm.org/10.1145/1513895.1513902

[Harbulot2004] Harbulot, B., Gurd, J., Using AspectJ to Separate Concerns in Parallel Scientific Java Code, In Proceedings of the 4th international conference on Aspect- oriented software development (AOSD '05 March 2004.

[Kephart2003] Kephart, J. O. and Chess, D. M.. The vision of autonomic computing. Computer, 36(1):41–50.

[Kephart2007] Kephart, J., Kephart, J., Chess, D., Boutilier, C., Das, R., Kephart, J. O., and Walsh, W. E.. An architectural blueprint for autonomic computing. Internet Computing, 18(21).

[Khronos2010] Khronos Group: The OpenCL Specification, v. 1.1 (September 2010), https://www.khronos.org/registry/cl/

[Khronos2011] Khronos WG. 2011. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl/. (Nov 2011).

[Kiczales1997] Kiczales, G., et al., 1997. Aspect-oriented programming. In ECOOP'97 — Object-Oriented Programming, M. AKŞIT and S. MATSUOKA Eds. Springer Berlin Heidelberg, 220-242.

[Kim2015] Youngsung Kim, Pavol Černý, and John Dennis. 2015. Performance search engine driven by prior knowledge of optimization. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2015). ACM, New York, NY, USA, 25-30.

[Mariani2011] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "ARTE: an Application-specific Runtime Management Framework for Multi-core Systems," In Proc. IEEE Symposium on Application Specific Processors (SASP'11), San Francisco, USA, June 2011, pp. 86-93.

[Martins2016] Luiz G. A. Martins, Ricardo Nobre, João M. P. Cardoso, Alexandre C. B. Delbem, Eduardo Marques: Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. TACO 13(1): 8 (2016)

[Martins2014] Luiz Martins, Ricardo Nobre, Alexandre Delbem, Eduardo Marques, and João M.P. Cardoso, "Exploration of Compiler Optimization Sequences using Clustering-Based Selection," in ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'14), June 12-13, 2014.

[Mayer2003] Mayer, W. and Stumptner, M.. Model-based debugging using multiple abstract models. In Proceedings of the 2003 International Workshop on Automated and Analysis-Driven Debugging, AADEBUG'03, pages 55–70.

[Medeiros2013] Medeiros, B., Sobral, J., "AOmpLib: An Aspect Library for Large-Scale Parallel Programming", in Proc 42nd International Conference on Parallel Processing (ICPP'13), Lyon, France, October 2013.

[Nobre2016] Ricardo Nobre, Luiz G. A. Martins, João M. P. Cardoso: A graph-based iterative compiler pass selection and phase ordering approach. LCTES 2016: 21-30.

[Nugteren2015] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC '15). IEEE Computer Society, Washington, DC, USA, 195-202. DOI=http://dx.doi.org/10.1109/MCSoC.2015.10

[Nvidia2010] Nvidia Corp.: NVIDIA CUDA C Programming Guide, v. 3.2 (2010), http:// developer.nvidia.com/object/gpucomputing.html

[OBoyle2013] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13). IEEE Computer Society, Washington, DC, USA, 1-10. DOI=http://dx.doi.org/10.1109/CGO.2013.6494993

[OpenACC2013] OpenACC. The OpenACCTMApplication Program Interface, August 2013. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf Version: 2.0a.

[OpenCL2008] The KhronosTMGroup. The Khronos Group Releases OpenCL 1.0 Specification. http://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification, December 2008. Accessed: 2014-02-07.

[Paone2015] E. Paone, F. Robino, G. Palermo, V. Zaccaria, I. Sander and C. Silvano. "Customization of OpenCL Applications for Efficient Task Mapping under Heterogeneous Platform Constraints" DATE 2015 – Design Automation and Test in Europe 2015.

[PGI2010] The Portland Group. PGI Accelerator Programming Model for Fortran & C, v1.3 (2010).

[Pinto2015] Pinto, P., Abreu, R., & Cardoso, J. M. (2015). Fault Detection in C Programs using Monitoring of Range Values: Preliminary Results. arXiv preprint arXiv:1505.01878.

[Popovici2002] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In Proc. 1st Int. Conf. on Aspect-oriented software development (AOSD '02). 2002, ACM, New York, NY, USA, 141-147.

[Psaier2011] Psaier, H. and Dustdar, S.. A survey on self-healing systems: Approaches and systems. Computing, 91(1):43–73.

[REFLECT10] REFLECT: Rendering FPGAs to Multi-Core Embedded Computing, Project Number 248976, FP7-ICT-2009-4, Activity: ICT-2009.3.6 Computing Systems. Duration in months: 36 (start: January 2010).

[Reyes2012a] Reyes, R., et al., 2012. accULL: An OpenACC Implementation with CUDA and OpenCL Support. In Euro-Par 2012 Parallel Processing, C. KAKLAMANIS, T. PAPATHEODOROU and P. SPIRAKIS Eds. Springer Berlin Heidelberg, 871-882.

[RL8-15] RL8 - Real Time Languages and tools for critical real time systems", NORTE-01-0124-FEDER-000062 (parte do projeto BEST CASE), 2013-2015.

[Santos2014] A. Santos, A DSL-based Approach for the Specification of Software Adaptations in Embedded Systems, PhD Thesis in Inf. Syst. and Comp. Eng., Univ. de Lisboa, IST, Lisboa, Portugal, 2014.

[Schmerl2006] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., & Yan, H.. Discovering architectures from running systems. IEEE Transactions on Software Engineering, 32(7), 454-466.

[Seo2013] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. 2013. Automatic OpenCL work-group size selection for multicore CPUs. In Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT '13). IEEE Press, Piscataway, NJ, USA, 387-398.

[Sobral2006] Sobral, J., Incrementally Developing Parallel Applications with AspectJ, IEEE IPDPS'06, Rhodes, Greece, April 2006.

[Suresh2015] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. 2015. Intercepting Functions for Memoization: A Case Study Using Transcendental Functions. *ACM Trans. Archit. Code Optim.* 12, 2, Article 18 (June 2015). DOI: http://dx.doi.org/10.1145/2751559

[Tan2010] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R., and Narasimhan, P.. Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In Proceedings of the 12th Network Operations and Management Symposium, NOMS, pages 112–119.

[Tiwari2009] A. Tiwari, C. Chen, J. Chame, M. Hall and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome.

[vanDeursen2000] A. van Deursen, P. Klint, and J. Visser, "Domain-specific Languages: An Annotated Bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000.

[Villazon2009] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. "Advanced runtime adaptation for Java," In Proc. 8th Int. Conf. on Generative Progr. and Component Eng. (GPCE '09). ACM, New York, NY, USA, pp. 85-94.

[Vos2000] Michael J. Voss and Rudolf Eigenmann. 2000. ADAPT: Automated De-Coupled Adaptive Program Transformation. In Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing (ICPP '00). IEEE Computer Society, Washington, DC, USA, 163-.

[Voss2001a] Michael J. Voss and Rudolf Eigemann. 2001. High-level adaptive program optimization with ADAPT. In Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming (PPoPP '01). ACM, New York, NY, USA, 93-102. DOI=http://dx.doi.org/10.1145/379539.379583

[Voss2001b] Michael Joseph Voss, "A generic framework for high-level adaptive program optimization," PhD Dissertation Purdue University, USA (2001).

[Wang2010] Wang, M., & Parashar, M. (2010, April). Object-oriented stream programming using aspects. In IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010 (pp. 1-11). IEEE.

[Wienke2011] Wienke, S., Plotnikov, D., an Mey, D., Bischof, C., Hardjosuwito, A., Gorgels, C., Brecher, C.: Simulation of bevel gear cutting with GPGPUs-performance and productivity. Computer Science - Research and Development 26, 165–174, 2011

[Wienke2012] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC: first experiences with real-world applications. In Proceedings of the 18th international conference on Parallel Processing (Euro-Par'12), Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer-Verlag, Berlin, Heidelberg, 859-870. DOI=10.1007/978-3-642-32820-6_85 http://dx.doi.org/10.1007/978-3-642-32820-6_85

[Wolfe2010] Michael Wolfe. 2010. Implementing the PGI Accelerator model. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU '10). ACM, New York, NY, USA, 43-50. DOI=10.1145/1735688.1735697 http://doi.acm.org/10.1145/1735688.1735697

[Yi2012a] Q. Yi, "POET: a scripting language for applying parameterized source-to-source program transformations," *Prog. Softw. Pract. Exper.*, vol. 42, no. 6, pp. 675–706, Jun. 2012.

[Yi2012b] Qing Yi. 2012. POET: a scripting language for applying parameterized source-to-source program transformations. Softw. Pract. Exper. 42, 6 (June 2012), 675-706.