Microprocessors and Microsystems 37 (2013) 1073–1089

Contents lists available at SciVerse ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Controlling a complete hardware synthesis toolchain with LARA aspects

João M.P. Cardoso [a,*], Tiago Carvalho [a], José G.F. Coutinho [b], Ricardo Nobre [a], Razvan Nane [d], Pedro C. Diniz [c], Zlatko Petrov [e], Wayne Luk [b], Koen Bertels [d]

[a] Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal INESC-TEC, Porto, Portugal
[b] Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, United Kingdom
[c] INESC-ID, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal
[d] Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands
[e] Honeywell International s.r.o, Turanka 100, 627 00 Brno, Czech Republic

ARTICLE INFO

Article history:
Available online 15 June 2013

Keywords:
Design-Space Exploration (DSE)
Automatic hardware synthesis
Hardware/software partitioning
Program transformations
Aspect-oriented programming
FPGAs

ABSTRACT

The synthesis and mapping of applications to configurable embedded systems is a notoriously complex process. Design-flows typically include tools that have a wide range of parameters which interact in very unpredictable ways, thus creating a large and complex design space. When exploring this space, designers must manage the interfaces between different tools and apply, often manually, a sequence of tool-specific transformations making design exploration extremely cumbersome and error-prone. This paper describes the use of techniques inspired by aspect-oriented technology and scripting languages for defining and exploring hardware compilation strategies. In particular, our approach allows developers to control all stages of a hardware/software compilation and synthesis toolchain: from code transformations and compiler optimizations to placement and routing for tuning the performance of application kernels. Our approach takes advantage of an integrated framework which provides a transparent and unified view over toolchains, their data output and the control of their execution. We illustrate the use of our approach when designing application-specific hardware architectures generated by a toolchain composed of high-level source-code transformation and synthesis tools. The results show the impact of various strategies when targeting custom hardware and expose the complexities in devising these strategies, hence highlighting the productivity benefits of this approach.

1. Introduction

The complexity of mapping applications expressed in high-level imperative programming languages to heterogeneous reconfigurable computing architectures (as is the case of FPGA-based architectures) is exacerbated by the variety of mapping tools and computing paradigms these architectures expose [1]. The best design solution for specific requirements, such as input data rates, may simply not be feasible for another set of requirements, such as the ones imposed by architectural constraints such as size, power or stage. Strict non-functional requirements (NFRs), such as reliability, safety, performance and energy consumption, are commonly out of the scope of existing tools or cannot be easily expressed using current high-level programming languages. Developing a feasible design that meets a set of competing, and often conflicting NFR is an extremely complex process.

The traditional mapping cycle targeting an embedded platform begins with an application description commonly used for validation in a software-only execution context (possibly executed in a General Purpose Processor – GPP). While aiming at specific target performance metrics, developers engage in global and localized code restructuring transformations, also bearing in mind the tools they have at hand. As such, starting from the application code, design-flows will lose complementary information about the computations, data-structures, and interfaces to other components involved in the system. This may lead to solutions that are only derived from a subset of the design-space. The use of specifications capturing this complementary information would make design-flows aware of important characteristics and may allow them to achieve more efficient implementations by early consideration of a variety of NFRs such as energy consumption, power dissipation, execution time, and fault-tolerance. One important issue related to complementary information is that it can be used to achieve specialized hardware/software implementations. This specialization may result from the knowledge of distinct properties such as possible value ranges of variables, loop iteration counts, and branch frequencies.

* Corresponding author.
E-mail addresses: jmpc@fe.up.pt (J.M.P. Cardoso), tiago.diogo.carvalho@fe.up.pt (T. Carvalho), gabriel.figueiredo@imperial.ac.uk (J.G.F. Coutinho), ricardo.nobre@fe.up.pt (R. Nobre), r.nane@tudelft.nl (R. Nane), pedro@esda.inesc-id.pt (P.C. Diniz), zlatko.petrov@honeywell.com (Z. Petrov), w.luk@imperial.ac.uk (W. Luk), k.l.m.bertels@tudelft.nl (K. Bertels).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.06.001

In addition, mapping applications to FPGA-based systems using current design practices requires developers to take into account various factors: to use and master a diverse set of tools, having to modify the application code according to the features supported by the tools in the design-flow, having to tune the application using code transformations, having to guide the toolchain in the design-flow according to the compiler optimization sequences, among other tasks [1]. Another issue contributing to an increased mapping complexity derives from the need to target the same application specification to distinct product lines, which is of paramount financial significance in terms of maintainability.

These factors lead to very long and error prone development processes, in practice forcing developers to settle for sub-optimal design solutions given the sheer size of the corresponding design-spaces. Even when *pragma*-based interfaces are provided, developers are confronted with the unpleasant prospect of having to annotate the source code. As the system evolves, these annotations and code specializations invariably lead to code obfuscation, and thus to designs that are difficult to port and maintain. Worse, they might simply become obsolete and interfere with existing and newer directives.

We have addressed these issues by proposing a novel synthesis toolchain for embedded systems which uses a methodology based on aspect-oriented programming (AOP) [2,3] concepts. This toolchain goes beyond the aforementioned issues related to hardware synthesis and spans also to compilation process and tools that are controlled and guided by aspect descriptions. The toolchain maps applications described in high-level imperative languages such as C to FPGA-based computing systems. For that purpose, the input application code is complemented with a LARA aspect-based language specification [4]. LARA allows developers to convey to the toolchain key input or domain-specific knowledge that can be exploited in the hardware and software tool-flows. Furthermore, LARA allows users to express compilation and synthesis strategies. These strategies can be very important as they may represent design patterns, compiler/synthesis sequences the user would like to experiment with, and/or the user may consider as good choices from previous design experiences. In addition, these sequences might be used to both reveal and/or prune certain design points when performing Design-Space Exploration (DSE).

The LARA aspect-based language [4] offers two significant benefits. First, it provides a uniform mechanism to convey to the underlying tools (using a common tool-independent interface) system attributes and NFRs [5] that are inaccessible in the current high-level programming paradigms. Second, by implementing an interface that decouples non-functional specifications from the application code, we preserve a clean functional description of the computations while taking advantage of a wealth of program and mapping transformations. One of the goals of LARA is to allow developers to specify compiler and synthesis strategies for hardware/software systems [6].

This paper makes the following specific contributions:

- It describes an automated hardware and software design-flow that combines tools from academia and industry. We have developed a weaver interface for each tool that allows them to be controlled from an external process.
- It describes the use of LARA, an aspect-based language used to control toolchain components via their weaver interfaces. LARA specifications allow developers to convey important non-functional application-specific requirements as well as compilation/synthesis strategies.
- It presents experimental results of the use of LARA strategies in the search of good loop unrolling factors and hardware loop pipelining. The case study presented here highlights the flexibility and tool interoperability in our aspect-based framework.
- It describes and presents experimental results about the control of the toolchain, including its back-end synthesis tools. This control mechanism allows outer-loop cycles over the toolchain. In particular we present LARA implementations of design-space exploration strategies involving the control and guidance of the different stages of the toolchain.

We see the flexibility of aspect-oriented approaches, such as LARA, as a key programming technology that will enable developers to meet increasingly demanding challenges in designing embedded systems. Our evaluation suggests that our approach offers a valuable mechanism to promote both performance and code portability while enhancing design reuse in the face of current and future architectures.

The remainder of this paper is structured as follows. In the next section we outline a possible LARA-based design-flow by providing a series of examples. These examples illustrate the structure and potential application of aspects in DSE. Section 3 describes a case study of the application of DSE to the mapping of a non-trivial kernel code to an FPGA device. In Section 4 we present experimental results of the use of LARA in the exploration of many hardware designs for this case study. In Section 5 we survey related work and then conclude in Section 6.

2. LARA-based design-flow

The REFLECT research project [7,8] aimed at supporting the multiple stages of mapping applications described in high-level programming languages such as C to multi-core embedded architectures. In particular, the project focused on the complete automation of the entire mapping process by building and evaluating the compiler/software toolchain, and providing a framework to address maintainability, verification and validation, as well as traceability issues when targeting these challenging platforms. We now provide an overview of the REFLECT toolchain, followed by a brief introduction of the key programming concepts for hardware synthesis using a domain-specific language, LARA, that relies extensively on aspect-oriented mechanisms.

2.1. Design-flow in REFLECT

The design-flow takes as input two types of specification as depicted in Fig. 1, namely:

- **Input Application**: The application's source code in an imperative procedural programming language such as C organized as one of more files.
- **LARA Specification**: The LARA descriptions capture non-functional requirements in the form of aspects and strategies. In particular, they define application characteristics such as precision representation, input data rates or even reliability requirements for the execution of specific code sections, as well as actions that guide the toolchain in an attempt to satisfy these requirements. In our context, a strategy defines a sequence of actions that are applied by compiler/synthesis tools to generate a specific hardware/software implementation.

The output of the toolchain is a complete hardware/software design. In the current implementation, the software component of the design is specified in source C files which can be compiled to the target processor using a native C compiler for that specific processor. The hardware component of the design is specified in RTL Verilog/VHDL and is used as input to synthesis tools such as Xilinx's ISE, which ultimately produces a bit-stream that can then be loaded onto an FPGA device for execution.

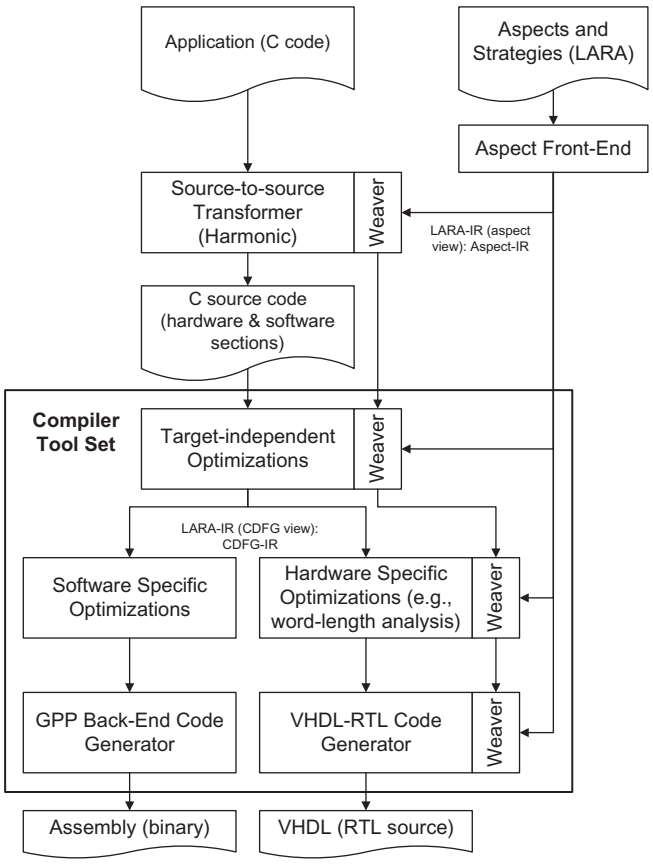

Fig. 1. The REFLECT project design-flow.

This design-flow is structured in three major components:

- **LARA Front-End**: The front-end converts LARA descriptions into Aspect-IR (Aspect Intermediate Representation) to be processed by the weavers which understand and process them as described below. The Aspect-IR is a low-level representation in XML format in which information is structured in a way to facilitate the parsing and interpretation of aspects and strategies.
- **Source-to-Source Weaver**: This stage performs, using an extended version of the Harmonic tool [9], source-level transformations (C to C) which include: arbitrary code instrumentation and monitoring, hardware/software partitioning using cost estimation models, as well as insertion of primitives to enable communication between software and hardware components. The result of this stage is a source file for each processing element reflecting each partition. Additional code is generated to implement remote procedure calls between the software and hardware partitions.
- **Compiler Tool Set**: This stage includes the front-end, middle-end and optimization phases of the input source-code compiler. The last two phases are common to both software and hardware partitions, which are target architecture independent. The back-end includes assembly code generators from software sections for the GPP, and VHDL/Verilog generators for specific hardware cores.

The REFLECT compiler tool set, named as *reflectc*, is based on the CoSy™ compiler framework [10] and integrates code generators for microprocessors and the DWARV 2.0 [11] hardware compiler (an enhanced CoSy-based version of the previous DWARV hardware compiler [12]), as VHDL generator. The *reflectc* compiler is thus based on a highly modular CoSy® design centered on a generic and extensible intermediate representation (IR), named as CCMIR [13,14], and integrating available CoSy code generators. An important component of *reflectc* is the weaver which interprets input LARA aspects and executes CoSy engines according to those aspect specifications. The CoSy weaver works at the CCMIR level.

One of the target architectures supported by *reflectc* is the Molen machine organization [15]. Molen is a reconfigurable architecture composed by three main elements: a shared memory and a General Purpose Processor (GPP) tightly coupled with Custom Computing Units (CCUs) and/or DSPs. The Molen architecture allows developers to exploit coarse-grained task-level parallelism by partitioning code sections that are computational intensive to dedicated hardware units (CCUs) while keeping control-intensive sections of an application in software (GPP). The DWARV 2.0 tool [11] is used to support the generation of these CCUs by translating C kernels to VHDL. In particular, DWARV 2.0 is a C-to-VHDL hardware compiler built with CoSy, and is composed by a set of engines that perform various standard transformations and optimizations on the input program at the CCMIR level.

In *reflectc*, the IR is generated by the C front-end CoSy standard engine. Then DWARV 2.0 engines perform both standard and hardware specific transformations and translate the control flow graph to a Finite State Machine (FSM), whereas the computations in each basic block are translated to hardware logic. For example, function variables become registers, function parameters are communicated through Molen eXchange registers [15], and local arrays can be mapped to both the FPGA (distributed) logic or to the local memories (implemented as FPGA BRAMs) directly connected to the CCUs. The current version of DWARV generates the VHDL representation of each CCU, including the interface used to communicate data and control information with the enclosing Molen machine organization.

To facilitate the integration of compilation and synthesis tools while providing a unified design-flow, we leverage the notion of weaver in our modular aspect-oriented framework. In particular, to integrate a new tool to our aspect-oriented design-flow, developers must implement a tool-specific interface to an existing LARA weaver, and/or must develop their own weaver with interface to the available LARA engine. This interface is responsible for communicating sequences of actions from the weaver to the tool, and attribute values and join points from the tool to the LARA weaver. Weavers can be seen as control engines that use the information captured in aspects to orchestrate the operation of corresponding tools. The current REFLECT design-flow includes two weavers, one integrated with the source-to-source transformer and the other incorporated in the compiler tool set. To show the capabilities and power of this mechanism for tool integration, we have conducted a number of experiments using alternative hardware compilers such as Catapult-C [16], which takes C source code and generates a Verilog design specification.

An additional benefit of the aspect-oriented approach lies in the fact that aspects can encapsulate a variety of strategies regarding the mapping, compilation or synthesis that provide the design-flow with the flexibility and modularity needed to derive combined hardware/software designs with desired characteristics or behavior. The experimental results presented in Section 4 clearly reveal this flexibility and productivity enhancements achieved by the current design-flow.

2.2. LARA aspects: goals, structure and examples

LARA is an aspect-oriented language geared towards hardware/software system design. LARA has been designed to capture non-functional requirements and to guide compilation and synthesis tools so that users can quickly develop design solutions that meet

these requirements, and which cannot be easily expressed using common programming languages such as C. In addition, LARA allows the definition of strategies as specifications that capture which aspects to apply and in what order. Ultimately, strategies can be seen as rules that implement specific hardware/software design patterns.

Fig. 2 depicts an aspect that maps a function named *filter_subband* to hardware to be synthesized for example on an FPGA device using a hardware synthesis tool. In this process, it invokes an aspect named "strategy1" which may include optimization rules, user knowledge, mapping strategies, target architecture properties, and other information specific to the function. This aspect also specifies two constraints related to input data ranges and noise power.

A second aspect, presented in Fig. 3, instructs the toolchain to fully unroll all innermost for-type loops in which the number of iterations is known at compile-time and does not exceed 32. LARA aspects can have input and output parameters. Input parameters give the possibility to execute the same aspect code with different values (e.g., loop unrolling factors). Since this aspect parameterizes the function name, it can be reused with other functions/applications as part of an optimizing strategy. In addition, users can increase its potential reuse by defining the number of iterations (32 in this example) and the unrolling factor as two additional aspect input parameters.

These two examples highlight the structure of an aspect definition. In addition to the name and possible parameters (the input section), an aspect includes three (3) main sections. A first section, the *select* section, defines the points (or artifacts) from the input program where an action is to take place. These points can correspond to statements, variables, and/or procedures in the source program. For each of these points there is a set of attributes that can be manipulated by an aspect. As an example, a loop can have as an attribute its type, control variable or even number of iterations. A second section, the *apply* section, specifies a sequence of actions to be performed at the selected set of points. It is thus an executable section and it is the responsibility of the programmer to ensure that these actions are consistent with the selected set of program artifacts. Lastly, the *condition* section, if not empty, defines under which conditions the *apply* section should be executed.

As an illustrative example, applying the aspect presented in Fig. 3 results in the code shown in Fig. 4 in which loop 1:1 has been fully unrolled.

As loops are important computational structures, we have provided in LARA a wide range of pre-defined attributes for loops. Loops in LARA can be identified by attributes such as *is_innermost*, *is_outermost*, and *rank* or *pragmas*. The *rank* attribute describes the relative position of loops in the code taking into account their possible nested structure. The identifiers of the loops in the example in Fig. 4 are defined by the following rank values: 1, 1:1, 2, and 2:1.

```
aspectdef maximizePerformance
  input funcName = "filter_subband"; end
  select  function{name==funcName}.arg{name=="s"} end
  apply $arg.noise_power <= 1E-3; end
  select  function{name==funcName}.arg{name=="z"} end
  apply $arg.range = "[-40..120]"; end
  select function{name==funcName} end
  apply
    $function.map(to:"processor", id:"virtex5");
    call strategy1();
    optimize(kind: "datarepr");
    call map2BRAMs(funcName);
  end
end
```

Fig. 2. Example of an aspect specifying non-functional requirements and mapping to hardware.

```
aspectdef strategy1
  input functionName end
  select function{name==functionName}.loop{type=="for"} end
  apply optimize(kind: "loopunroll", k:"full"); end
  condition $loop.numIterIsConstant &&
            $loop. num_iter <= 32 && $loop.is_innermost
  end
end
```

Fig. 3. Aspect module for fully unrolling innermost loops with a number of iterations no greater than 32.

Fig. 5 shows three aspect versions that perform loop unrolling on four (4) loops from the *filter_subband* example (Fig. 4) according to individual loop unrolling factors passed as parameters to the aspect. The first version (Fig. 5a) considers 4 input parameters ($k_1,\ldots,k_4$) as unroll factors. The second version (Fig. 5b) considers an array of factors as input. Both these two versions are very specific to the code of the *filter_subband* example as they both use rank values referring to the loops in the example. The third version (Fig. 5c) uses a reusable aspect that receives an array of loop unrolling factors as the input argument, and performs loop unrolling for all the for-type loops according to those factors. This example considers that each factor in the array is associated to a specific loop by the order the loops are woven (in this example: $1 \rightarrow 1{:}1 \rightarrow 2 \rightarrow 2{:}1$).

For instance, using this approach, the *reflectc* compiler can be executed over the code of the *filter_subband* function using the LARA aspect and specifying the inputs of the aspect in the command line to produce C, assembly, or VHDL code. This allows users to easily try different combinations of the loop unrolling factors.

2.3. Design-space exploration in LARA

We have extended the LARA language to provide an outer-loop design-flow mechanism for toolchains as depicted in Fig. 6. In our outer-loop approach, aspects can capture design-space exploration strategies by controlling single or multiple executions of tools in a design-flow. Moreover, iterations of the toolchain can be codified allowing the toolchain to repeat execution with possibly different parameters while certain conditions are satisfied. Outer-loop aspects can be captured by LARA scripts to exploit the semantics of the *select-apply-condition* sections [4], including the use of imperative code in *apply* sections. By extending the portfolio of executable commands and feedback report mechanisms, the *apply* section of a LARA aspect can support the definition and implementation of DSE schemes. The LARA outer-loop mechanism can be applied to LARA-based toolchains (as the ones represented in Fig. 6) or to other toolchains.

The LARA outer-loop is responsible for executing the tools in the toolchain and to acquire the information provided by these tools. This information can then be used to decide which strategy to be applied and/or the values of parameters to be used by these strategies. The decisions may need calculations and or DSE algorithms which can all be specified in LARA. DSE outer-loop schemes expressed in LARA may include executing the toolchain multiple times (e.g., a pre-defined number of times or while some condition applies).

The LARA outer-loop interpreter, named **larai**, is a LARA weaver with core interpreter engine based on Rhino [17], an open-source implementation of a JavaScript interpreter written entirely in Java. The main objective of **larai** is to provide an outer-loop (external) aspect-oriented mechanism to control the components of a

```
1  void filter_subband(double z[512], double s[32], double m[32][64]) {
2    double y[64];
3    int i,j;
4    for (i=0;i<64;i++) { // loop 1
5      y[i] = 0;
6      for (j=0; j<8;j++)   // loop 1:1
7        y[i] += z[i+64*j];
8    }
9    for (i=0;i<32;i++) {  // loop 2
10     s[i]= 0;
11     for (j=0;j<64;j++)   // loop 2:1
12       s[i] += m[i][j] * y[j];
13   }
14 }
```

Fig. 4. Code of the *filter_subband* function. Loop 1:1 is fully unrolled when using the aspect shown in Fig. 3.

(a)

```
aspectdef UnrollLoopsByRank
  input k1, k2, k3, k4 end
  select function.loop end
  apply
    if($loop.rank == "1") {
      optimize(kind:"unroll", k:k1);
    } else if($loop.rank == "1:1") {
      optimize(kind:"unroll", k:k2);
    } else if($loop.rank == "2") {
      optimize(kind:"unroll", k:k3);
    } else if($loop.rank == "2:1") {
      optimize(kind:"unroll", k:k4);
    }
  end
end
```

(b)

```
aspectdef UnrollLoopsByRank
  input factors end
  select function.loop end
  apply
    if($loop.rank == "1") {
      optimize(kind:"unroll", k:factors[0]);
    } else if($loop.rank == "1:1") {
      optimize(kind:"unroll", k: factors[1]);
    } else if($loop.rank == "2") {
      optimize(kind:"unroll", k: factors[2]);
    } else if($loop.rank == "2:1") {
      optimize(kind:"unroll", k: factors[3]);
    }
  end
end
```

(c)

```
aspectdef UnrollLoopsByRank
  input factors end
  var i = 0; // index to factors
  select function.loop{type==”for”} end
  apply
    optimize(kind:"unroll", k:factors[i++]);
  end
end
```

Fig. 5. Aspect module for loop unrolling considering four (4) for-loops from the *filter_subband* example: (a) using 4 input parameters, one per loop; (b) using an input array with the unrolling factors instead of 4 parameters; and (c) using an input array with the unrolling factors and assuming the ordering given by the order of the join points.

toolchain. LARA aspects input to **larai** can include instructions to execute tools, explore configurations and/or command line options, get attribute values from reports, and decide whether to continue to explore different configurations and/or options based on the results achieved at a particular stage of the design-flow.

Fig. 6 shows how the LARA outer-loop interpreter is integrated with the REFLECT toolchain. Note, however, that the LARA outer-loop can be easily coupled with other toolchains and the feedback mechanism using information obtained from tool reports is the only extension required.

The LARA outer-loop design-flow mechanism can be used at different levels of the toolchain. This can be in fact a useful strategy to DSE as the user may start exploring the components of the toolchain at a higher-level where decisions and options can be evaluated faster, and then explore the lower parts of the toolchain where feedback information is more accurate, but where design points are evaluated more slowly.

The LARA outer-loop interpreter (**larai**) receives the report information through a global object identified by the *attributes* (or by the abbreviation: @) variable. It can then easily access reported values, reassign values to existent attributes, or even add new attributes and values. Fig. 7 shows a LARA code which prints the value of a specific attribute of function “f1” (the value of that attribute is accessed by the LARA code: *@function{“f1”}.cost_virtex5*) and adds a new attribute to the same function (using the LARA statement: *@function{“f1”}.new_attribute = “value”;*).

As illustrated in Figs. 8 and 9, developers can explore different mapping parameters while exercising the execution of specific tools. In these examples, the aspects define search algorithms for the most effective loop unrolling factors and loop pipelining initiation intervals. At each step of this search, the LARA engine (through its weaver) invokes, and observes the output metrics of the hardware design derived by the use of a given set of transformations. The weaver first invokes the optimization of a source code representation (via the *run* command) followed by the invocation of a high-level synthesis tool (via the *run* command). It then observes resulting design metrics, such as “Latency” and “maxFreq”, exposed internally to the LARA code. The best hardware design is then selected using a specific metric of efficiency.

These two examples highlight the modularity of our LARA-based approach. Two external LARA aspects, *simul* and *xilinx*, are called in the examples. Those aspects include the LARA code to execute the ModelSim simulator and the Xilinx ISE™ tools and are thus reusable aspects.

We have developed an API that allows LARA code and **larai** to control and execute third-party tools such as the Xilinx ISE tools. The LARA outer-loop environment uses this Java API and its methods to run each tool in command line mode [18], to pass execution options, and to extract the information from the output report files. With this interface the LARA strategies can control the execution and extract important information from reports output by Xilinx ISE tools (such as *xst* [19], *map*, *par*, and *xpwr* [20]). Fig. 10, shown below, provides an excerpt of LARA code defining values for the input parameters for the Xilinx *xst* tool, and the execution of the Xilinx tools: *xst*, *ngdbuild*, *map*, *par*, and *xpwr*. Note that the input parameters of the other tools, besides *xst*, can be defined in LARA in a similar way.

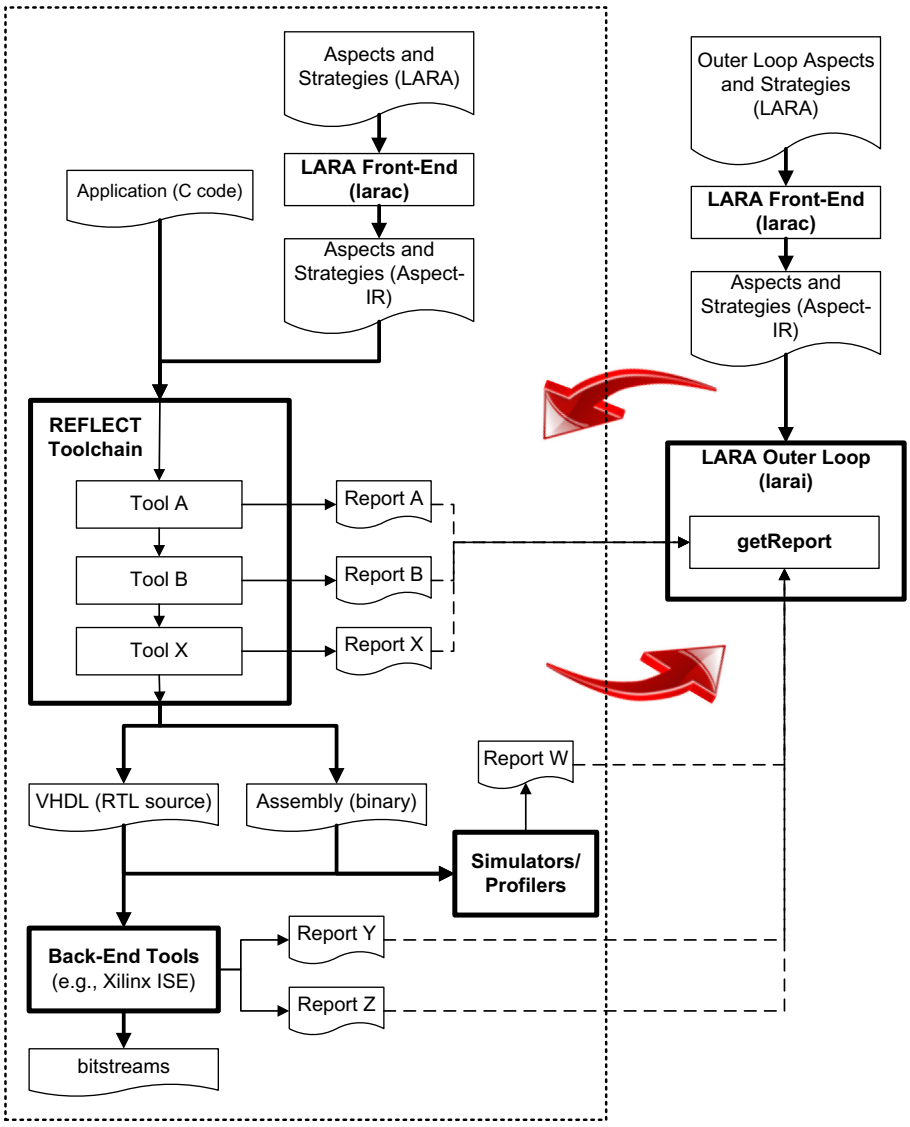

Fig. 6. LARA-based design-flow considering the LARA outer-loop mechanism for controlling and iterating over toolchains (in this case, we show a LARA-based toolchain). The two large arrows represent outer-loop actions and feedback information to/from the LARA-based toolchain.

```
attributes.set({
    function:{
      f1:{
         cost_virtex5: 13245,
          cost_ppc: 21137,
      }
    }
});
```

```
aspectdef MonitoringAttributes
    ...
    println("Virtex cost: "+@function{"f1"}.cost_virtex5); // print attribute cost_virtex5
    @function{"f1"}.new_attribute = "value"; // define a new attribute and assign it to a value
    ...
end
```

Fig. 7. Using attributes: (a) code example used to communicate values of attributes (returned by tools or extracted from reports) to the LARA outer-loop and (b) code of an aspect using attributes.

The Java code methods extract the information from the reports and return excerpts of JavaScript code (such as the one given in Fig. 11) which are then run by **larai** and exposed to the LARA code through the global variable *attributes*. This process makes our approach very easy to extend and allows a smooth integration of other tools.

In order to have a feedback loop from simulations of the generated HDL (Hardware Description Language) code, we provide in our Java report API a method that extracts the number of clock cycles and the final status of the simulation (whether it failed or not). This information is extracted from the report produced by ModelSim™ after simulating the generated VHDL code.

3. Case study: Aspects in action

We now describe the application of optimization and communication mapping aspects presented in the previous section. In this case study, we focus on *filter_subband*, a critical function of the

```
import simul; // external LARA aspect

aspectdef ExploreLoopUnrolling
 input
   factorsi = [1, 2, 4, 8, 16, 32, 0]; factorsj = [1, 2, 4, 0];
   factorsl = [1, 2, 4, 8, 16, 0]; factorsk = [1, 2, 4, 8, 16, 32, 0];
 end

 // evaluate loop unrolling
 for(var i=0; i<factorsi.length; i++) {
   for(var j=0; j< factorsj.length; j++) {
     for(var l=0; l< factorsl.length; l++) {
       for(var k=0; k< factorsk.length; k++) {
          run(tool: "reflectc", file: "filter_subband.c", gen:"vhdl", asp: "UnrollLoopsByRanks",
                  param: "("+factorsi[i]+","+ factorsj[j]+","+factorsl[l]+","+factorsk[k]+")");
          call simul(design: "filter_subband");
          println("#clock cycles: "+@design.filter_subband.Latency);
       }
     }
   }
 }
end
```

Fig. 8. Example of a LARA aspect exploring designs resulting from distinct values of loop unrolling factors for 4 loops.

```
import simul; // external LARA aspect
import xilinx; // external LARA aspect

aspectdef ExploreLoopPipelining
  input function_name = "filter_subband" end

  var iiVals = [1, 2, 3, 4, 5, 8, 12, 16]; // initiation interval (II) values to be explored

 var execTime = Number.MAX_VALUE;
 var newExecTime;
 var selectedII;
 for(var i =0; i< iiVals.length; i++) {
   var ii = iiVals[i];
   run(tool: "harmonic", name: function_name, opts:{ opt: "looppipelining", II: ii});
   run(tool: "hls-tool", name: function_name);
   call simul(name: function_name);  //call to other aspect
   call xilinx(tool:"xst", name: function_name); // call to other aspect
   newExecTime = @function[function_name].Latency / @function[function_name].maxFreq;
   if(newExecTime < execTime) {
      execTime = newExecTime;
      selectedII = ii;
   }
 }
 println("Selected initiation interval (II): "+selectedII);
end
```

Fig. 9. Example of a LARA aspect exploring designs resulting from distinct values of pipelining initiation intervals.

MPEG Audio Encoder (MPEG-2 Layers I and II) application. Fig. 4 presents the C code implementation of this function, which is used in the Polyphase Filter Bank, a key component of the encoder. This function receives 512 audio samples and outputs 32 equal-width frequency sub-bands. The C code is structured as four (4) for-type loops computing arithmetic convolutions and accumulations using, in one instance, filter coefficients stored in array variable *m*.

This function offers a wide range of transformations and thus optimization opportunities at the loop level, namely:

- To transform the double-precision floating-point data types used to single-precision floating-point or fixed-point representations.
- To apply, given constant loop bounds, strength-reduction to array index calculations and use small bit-width representations for loop control variables.
- To unroll loops in order to expose vast amounts of instruction-level-parallelism (ILP) leveraged by concurrent hardware adder and multiplier blocks.
- To privatize the storage associated with the *y* array in local RAM modules. The values of this array are written in the first loop nest (Fig. 4, lines 4–8) and read in the section loop nest (Fig. 4, lines 9–13). This transformation will reduce the number of load and store operations by replacing them with internal memory read and write operations.
- To replace accumulation variables, such *y* and *s* arrays, to scalar variables in order to promote them to registers.

```
var DesignName = "f1";
var xstOpts = { // specify xst options and values
    "-opt_mode": "Area", "-opt_level": "1", "-power": "NO", "-keep_hierarchy": "NO", "-fsm_encoding": "Auto",
    "-fsm_style": "LUT", "-ram_extract": "Yes", "-ram_style": "Auto" , "-rom_extract": "Yes",
    "-mux_style": "Auto","-decoder_extract": "YES", "-priority_extract": "YES",    "-shreg_extract": "YES",
    "-shift_extract": "YES", "-xor_collapse": "YES", "-rom_style": "Auto", "-auto_bram_packing": "NO",
    "-mux_extract": "YES","-resource_sharing": "YES", "-use_dsp48": "Auto", "-register_duplication": "YES",
    "-register_balancing": "NO", "-slice_packing": "YES", "-optimize_primitives": "NO",
    "-equivalent_register_removal": "YES",  "-slice_utilization_ratio_maxmargin": "5"
 };
run(tool: "xst", args: {design: DesignName, folder: "VHDL-project", xstOptions: xstOpts});
run(tool: "ngdbuild", args: {design: DesignName});
run(tool: "map", args: {design: DesignName});
run(tool: "par", args: {design: DesignName});
run(tool: "xpwr", args: {design: DesignName});
```

Fig. 10. Example of LARA code to control the Xilinx back-end tools.

```
attributes.set({
    design:{
        f1:{
            msgWarnings:0,
            device:"5vlx50tff1136",
            msgInfos:0,
            delay:20.167,
            numSliceLUTs:597,
            msgErrors:0,
        }
    }
});
```

Fig. 11. Code with attribute values extracted from a Xilinx *xst* report.

- To apply, for each loop, e.g. in both loop nests and for both inner and outer loops, hardware pipelining execution when offered as one of the implementation choices of the high-level synthesis (HLS) tool at hand.

We address here the use of LARA strategies to improve the performance of the *filter_subband* function (Fig. 4). One possible strategy is to unroll 2 × loop 1 and to jam the resulting two instances of loop 1.1, and to unroll 2 × loop 2.1. Fig. 12 depicts the code considering that strategy, which results from the transformations specified by the two aspects shown in Fig. 13. These aspects include two options (A and B) for identifying the two loops to be merged (fused/jammed). Option A explicitly identifies the loops by using the *rank* attribute. Option B identifies the first loop using the *rank* attribute and considers that fusion is performed considering both loops in the same hierarchical level.

The first aspect (*filtersubbandStrategy1*) presented in Fig. 13 starts by defining a variable *functionName* with the name *filter_subband* (line 2). In line 3, a *select* expression is responsible to select all the for-type loops in the code of the function with name given by the *functionName* variable (i.e., *filter_subband*). This corresponds to the four (4) for-type loops in the code in Fig. 4. For all these loops, two optimizations are invoked: *loopanalysis* and *loopscalar* (see *apply* section in line 4). The *apply* section in line 5 also specifies actions to be applied to the same loops given by the select section in line 3. However, in this latter case a condition section (line 6) is able to filter the loops that will be affected by the *loopunroll* optimization (with a factor $k = 2$). In this case, only the loop in *rank* 1 (loop in line 4 of Fig. 4) will be affected. The other loop to be unrolled twice is the loop corresponding to *rank* 2:1 (see lines 11 and 12 of Fig. 4). Line 12 of Fig. 13 has an apply section which invokes the aspect *loopjam* passing as arguments the value of *func-*

```
1   void filter_subband(float z[512], float s[32], float m[32][64]) {
2     ...
3     for (i=0;i<64; i+=2) {
4       y_aux1 = 0.0; y_aux2 = 0.0;
5       for (j=0, j<8;j++)  {
6         y_aux1 += z[i+64*j];
7         y_aux2 += z[i+1+64*j];
8       }
9       y[i] = y_aux1; y[i+1] = y_aux2;
10    }
11    for (i=0;i<32;i++) {
12      s_aux= 0.0;
13      for (j=0, j<64; j+=2) {
14        s_aux +=  m[i][j] * y[j];
15        s_aux +=  m[i][j+1] * y[j+1];
16      }
17      s[i] = s_aux;
18    }
19  }
```

Fig. 12. Code after double to float conversion, unroll and jam (first set of loops) and unroll 2× (innermost loop in the second set of loops).

tionName and 1:1 and 1:2 (the actual *ranks* of the loops to be fused).

The two options of the aspect *loopjam* are represented in lines 14–21 and 24–29. The first option (Option A) uses the *rank* attribute (1:1 and 1:2) in the input source code to identify the loops. The second option (Option B) only uses the *rank* attribute for the original loop nest (i.e., loop in line 6 of Fig. 4) and assumes that the optimization will fuse this loop with the instance of the loop obtained by loop unrolling by two the outermost loop (i.e., loop in line 4 of Fig. 4).

Instead of this indexing scheme with the *rank* attribute to identify loops in the code, one can use annotations placed in the original C code. Fig. 14 shows the *filter_subband* function of Fig. 4, but with annotations (using pragmas) to identify the three original loops that will be affected by the strategy. Fig. 15 shows the strategy defined above, but now with the identification of the loops using the annotations presented in Fig. 14.

Fig. 16 shows a strategy defined with a LARA script that repeats the application of compiler optimizations while changes are observed. The objective is to fully unroll all for-type loops which exhibit the following properties: the number of iterations is known at compile time and is less or equal than 20 and does not contain inner loops. The strategy starts from the innermost loop outwards and iteratively traverses the loop hierarchy.

```
1  aspectdef filtersubbandStrategy1
2      var functionName = "filter_subband";
3      select function{name==functionName}.loop{type=="for"} end
4      apply optimize("loopanalysis"); optimize("loopscalar"); end
5      apply optimize(opt: "loopunroll", k: 2); end
6      condition $loop.position == "1" end
7      apply optimize(opt: "loopunroll", k: 2); end
8      condition $loop. rank == "2:1" end
9      // needs a call to an aspect as the two loops to jam are the
10     //result from the apply in line 6 and their join points
11     // are not visible in this aspect
12     apply call loopjam(functionName, "1.1","1.2"); end
13 end                                                    Option A
14 aspectdef loopjam
15     input funcName, pos1, pos2 end
16     L1: select function{funcName}.
17         ($l1=loop){(type=="for", rank ==pos1)} end
18     L2: select function{funcName}.
19         ($l2=loop){(type=="for", rank ==pos2)} end
20     apply to L1::L2 $l1.optimize loopfusion($l2); end
21 end
22     apply call loopjam(functionName, "1.1"); end
23 end                                                    Option B
24 aspectdef loopjam
25     input funcName, looprank end
26     select function{name==funcName}.
27         loop{(type=="for", rank == looprank)} end
28     apply $loop.optimize("loopfusion"); end
29 end
```

Fig. 13. Aspects defining a strategy referring loops with the indexing scheme using the *rank* attribute.

```
1  aspectdef filtersubbandStrategy1
2      var functionName = "filter_subband";
3      select function{name==functionName }.loop{type=="for"} end
4      apply optimize("loopanalysis"); optimize("loopscalar"); end
5      apply optimize(opt: "loopunroll", k: 2); end
6      condition $loop.name == "for1" end
7      apply optimize(opt: "loopunroll", k: 2); end
8      condition $loop.name == "for4" end
9      // needs a call to an aspect as the two loops to jam
10     // are the ones resultant from the apply in line 6 and
11     // their join points are not visible in this aspect
12     apply call loopjam(functionName, "for1","for2"); end
13 end
14 aspectdef loopjam                                     Option A
15     input funcName, loopname1, loopname2 end
16     L1: select function{name==funcName}.
17         ($l1=loop){(type=="for", name==loopname1)} end
18     L2: select function{name==funcName}.
19         ($l2=loop){(type=="for", name==loopname2)} end
20     apply to L1::L2 $l1.optimize("loopfusion",$l2); end
21 end
22     apply call loopjam(functionName, "loop1"); end
23 end
24 aspectdef loopjam
25     input funcName, loopname end
26     select function{name==funcName}.                 Option B
27         loop{(type=="for", name==loopname)} end
28     apply $loop.optimize("loopfusion"); end
29 end
```

Fig. 15. Aspects defining a strategy for referring to loops by names defined in the annotations.

In addition to these code transformations, one could also exploit transformations and implementation optimizations related to arithmetic representations and operators. Using aspects, developers can specify the data type and scaling of the accumulator for fixed precision arithmetic. In addition, aspects can also leverage domain-specific knowledge indicating data rates associated with specific variables and thus imposing latency requirements to generate feasible design solutions.

All these transformations can lead to very large design spaces and thus beyond a reasonable manual exploration. In Fig. 17 we illustrate how we can extend the strategy presented in Fig. 4 to support a wide range of options based on a set of optimizations (aspect definitions that drive a particular transformation) parameterized via a set of arguments, which are passed down to each optimization. More complex and sophisticated search strategies can be defined in LARA so that they can leverage an existing set of transformational aspects.

It can be seen that LARA provides powerful integration mechanisms which allows developers to easily code a DSE strategy, e.g., as the one using an approach similar to the DSE strategy proposed in [22]. Fig. 18 shows the code section of a LARA strategy that contains a rule to decide whether to execute low-level hardware synthesis tools (Xilinx *xst* [19], in this case). This strategy also gives strong evidence of the sophistication provided by LARA and its integrated environment.

```
...
 #pragma name="for1"
  for (i=0;i<64;i++) {
     y[i] = 0;
     #pragma name="for2"
     for (j=0; j<8;j++) ...
  ...
  for (i=0;i<32;i++) {
     s[i]= 0;
    #pragma name="for4"
     for (j=0;j<64;j++)  ...
...
```

Fig. 14. Use of annotations (in the form of C pragmas) to identify specific loops.

4. Experimental results

This section presents experimental results using various hardware/software transformations specified by mapping strategies using aspects as described in the previous section. In particular, we make use of the aspects described in Sections 2 and 3 for the *filter_subband* kernel code. We target a Xilinx Virtex-5 (5vlx50tff1136) FPGA for our hardware designs using *reflectc* (with embedded DWARV 2.0 [11] VHDL generation engines) and Xilinx ISE 12.2 synthesis, mapping and placement and routing tools (*P&R*). Note, however, that our approach is tool agnostic and is capable of supporting other tools, such as Catapult-C [16] and Mentor Graphics Precision 2010a synthesis tool [21], as well as the Xilinx ISE placement and routing tools, as previously reported in [23]. The latencies and execution times for each hardware design presented in this section are related to the execution of the CCUs (hardware units) generated by *reflectc*. The tools were executed in a machine with an Intel®Xeon®CPU E7330, @2.40 GHz, with 2 GB de RAM and with a Linux-based operating system.

The design experiments presented here are organized in two sets. The first set of experiments corresponds to the applications of LARA strategies for loop unrolling in conjunction with other compiler optimizations such as scalar replacement (identified here as "loopscalar" as it is an optimization applied in the context of loops). In this set of designs, we use aspect strategies that lead to hardware implementations of the transformed codes exploring various loop unrolling factors for distinct loops in the original code. The second set of experiments considers LARA strategies with other loop-based code transformations, namely, loop unrolling, loop merge (also known as fuse or jam), scalar replacement, and

```
1  aspectdef Strategy2
2      input functionName="f1" end
3      select function{name==functionName} end
4      apply
5          do {
6              call A:repeatloopunroll(functionName)
7          } while(A.change); // the use of a default output attribute
8      end
9  end
10
11 aspectdef  repeatloopunroll
12     input functionName end
13     select function{name==functionName}.loop{type=="for"} end
14     apply
15         optimize("loopanalysis");
16         optimize("loopscalar");
17         optimize("loopunroll","full"); // fully unroll loop
18     end
19     condition $loop.is_innermost && $loop.numIterIsConstant && $loop.num_iter<=20; end
20 end
```

Fig. 16. A LARA strategy for performing optimizations while actions produce changes.

```
aspectdef ExploreStrategies
 input // input parameters with default values
   function_name = "filter_subband"
   optimizations  = [pipeline, unroll, s1, s2];
   args = [[[1],[8],[16]], [[2],[3]], [  ], [ [1,2] ] ];
   clock_freq = 200;
 end
 select function{function_name} end
 apply
  var exec_time = Number.MAX_VALUE;
  var new_exec_time;
   /* opts=> [[pipeline,[1]], [pipeline,[8]], ..., [s2,[1,2]]] */
  opts = combine(optimizations , args);
  for(var i=0; I < opts.length; i++) {
    opts [i][0] (opts [i][1]);
    run(tool: "harmonic",  name: function_name, opts:{opt:"pipelining", opts:  opts [i][0] (opts [i][1])});
    run(tool: "hls-tool", name:function_name);
    report(tool: "hls-tool");
    new_exec_time = @design[function_name]. latency / clock_freq;
    if(new_exec_time < exec_time) {
     exec_time = new_exec_time;
     strategyID = i;
    }
  }
  println("Selected strategy id: "+strategyID);
 end
end
```

Fig. 17. Example of a LARA aspect for exploring different optimizations and corresponding parameters values.

combinations thereof (e.g., unroll and jam). Although these transformations are not exclusively used in hardware synthesis, they are traditionally viewed as well suited for this purpose.

4.1. Exploring loop unrolling factors

To explore a large number of designs corresponding to different loop unroll factors (power-of-two values) for the two innermost loops of *filter_subband*, using the outer-loop LARA strategy mechanism described in Section 2.3. With this strategy, the toolchain automatically derives a VHDL specification of the input source code computation and invokes ModelSim for simulating each design point. This code generation and simulation required about 10 min for a total of 28 distinct design points. The outer-loop LARA aspect also uses the maximum clock frequency estimated using Xilinx XST (*xst*) [19] after the simulation of each design point to determine the expected time performance of each design. This combined execution of the *xst* tool required approximately 62 min (and 2.4 h when considering also placement and routing) when using a low level effort parameter setting.

Figs. 19–21 show the results obtained with two LARA strategies when coupled with the outer-loop LARA aspect. The design points are identified by the loop unrolling strategies. Table 1 shows some examples of design points and their meaning. These design points consider different loop unrolling factors for the two innermost loops (i.e., loops 1.1 and 2.1) of the *filter_subband* function in Fig. 4.

The design with the lowest number of clock cycles corresponds to the full loop unrolling of the two (2) innermost loops in *filter_subband* (Design Point 0:00). It achieves a 25.3% reduction of clock cycles when compared to the design without loop unrolling

```
...
aspectdef ExploreStrategies
  ...
  // loop defining loop unrolling factors
  // inner loop:
        call A:CompileWithFactors(factors);
        call B:simul("filter_subband");
        latency = B.Latency;

        // rule to decide to execute hw synthesis
        if(latency < factor*BestDPMinLatency) {
             call xilinx(tool: "xst", design:"filter_subband");
             MaxFreq = @design.CCU.maxFreq;
             RefFreq = MaxFreq;
             ExecT = (latency/MaxFreq);
             if(ExecT < BestDPExecT) {
                BestDPMinLatency = latency;
                BestDPMaxFreq = MaxFreq;
                BestDPExectT = ExecT;
                BestDesignPoint = factors.toString();
             }
          }
  // end loop defining loop unrolling factors
  ...
end
```

Fig. 18. Example of a LARA aspect which considers low level toolchain stages.

(Design Point 1:01). Still, the fluctuations of maximum clock frequencies according to the designs should be taken into account when considering that systems will operate at close to maximum clock frequencies. For the 28 designs, there is a maximum reduction of near 26% from the highest and the lowest clock frequency. When considering the maximum clock frequencies estimated by *xst*, the design with best performance is the one considering full unrolling of the first innermost loop and unrolling 8× the second innermost loop (Design Point 0:08). This design achieves a 26.7% reduction on execution time when compared to the design without loop unrolling (Design Point 1:01).

LARA also provides an integrated environment that allows strategies to exploit data derived from multiple tools without an extra programming effort as is the case with estimation and simulation tools. In order to avoid longer runtimes involved in executing low level tools for each of the design points being evaluated, we easily changed our strategy to explore loop unrolling factors and consider only low-level toolchain stages under certain conditions. We use a rule to decide, after the simulation with ModelSim, the execution of the hardware synthesis tools to estimate the maximum clock frequency. In particular, the rule uses the condition $latencyCurrentDesign < factor * bestLatency$ to make that decision (see the LARA code section presented in Fig. 18). Our strategy for the clock frequency of the design points to which no *xst* execution was performed (and thus, no estimation of the clock frequency is available) was to use the previous clock frequency estimated by *xst*. This way we reduce the time associated to the execution of the DSE by avoiding the execution of *xst* for many design points. Note that revising this strategy, which had a significant impact in the design-space exploration, involved modifying a single script and was performed in a few minutes. This highlights the potential of LARA and its integrated environment.

With this approach, we explored the 28 design options/points with a reduction of 13% of the time elapsed for running the DSE. When using a *factor* equal to one, *xst* was executed for 17 of the 28 designs (a reduction in the number of *xst* executions of 33%). When considering placement and routing we obtained a reduction of 22.3% of the time elapsed for running the DSE. Note that LARA strategies considering different toolchain stages with different levels of estimations (from high-level to low-level estimations), and sophisticated heuristics and/or iterative optimization schemes (such as Simulated Annealing) are easy to program in our approach.

Table 2 shows the number of design points considered and the different execution times to run the experiments when applying the two strategies, i.e., the first one considering that each stage of the toolchain is executed over all of the 28 design points, and the second one considering an approach that skips some of the executions of the back-end toolchain stages (*xst* and *P&R*).

Fig. 22 shows the results obtained with our simple strategy. When equals to 1, the line "level" represents the execution of the *xst* tool to estimate the maximum clock frequency. The value of 0 means that *xst* was not executed for the respective design point and the clock frequency used was the one estimated by the previous *xst* execution.[1] The line "Normalized Exec. Time (xst clkFreq)" shows the results obtained during DSE and when estimating via *xst* the maximum clock frequency for each design. The line "Normalized Exec. Time (prev. clkFreq)" shows the results obtained during DSE and using the strategy that avoids the execution of low-level (back-end) stages for every design point (DSE-skip strategy). With this DSE-skip strategy we were able to avoid the execution of *xst* for 7 of the 28 design points and to identify a design (Design Point 0:16) which is 1.66% below the best performance obtained (Design Point 0:08) when considering the execution of *xst* for each design point evaluated.

4.2. Exploring more advanced LARA strategies

In the second set of experiments, we consider the application of additional LARA strategies to the *filter_subband* function. Table 3 presents the optimizations considered for each LARA strategy. We evaluate each of the 12 designs obtained by using simulation (with ModelSim) and synthesis (with *xst*).

Fig. 23 illustrates the results achieved in these experiments. Design 10 exhibits the minimum execution time and it allows a 79% reduction (a speedup of 4.8×) over the base implementation (Design 0) for *filter_subband*. Fig. 24 shows the hardware resources considering FPGA LUTs and registers. With respect to the base design (Design 0), the design with the lowest execution time (Design 10) requires about 4.9× and 7.5× more registers and LUTs, respectively. Note that for this set of optimizations, a strategy as the one experimented in Section 4.1 to reduce the number of hardware synthesis (*xst*) executions was able to identify the design with the best performance.

4.3. Summary

In summary, LARA and the REFLECT integrated environment provide a framework for assisting developers in programming and evaluating DSE strategies, such as sequences of compiler optimizations and synthesis options. The integrated view allows developers to use the same environment to explore from source code transformations, passing through compiler optimizations, to hardware synthesis parameters as the ones in the Xilinx ISE tools such as *xst*, *map*, and *par*. Besides other benefits of the LARA approach (see, e.g., [4–6]), we believe that the features described in this article make the approach very useful for the community developing hardware/software systems.

[1] Note that this is a simple heuristic used and the main point here is to show that programming this type of DSE schemes is easy using LARA and its integrated environment.

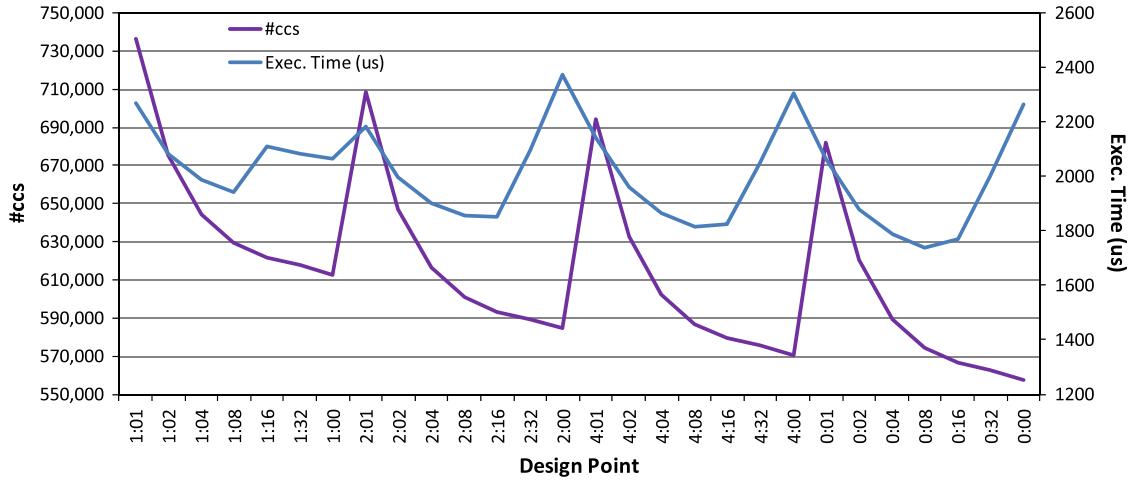

Fig. 19. Experimental results related to the number of clock cycles and execution time for the *filter_subband* function considering unrolling of the innermost loops by different unrolling factors.

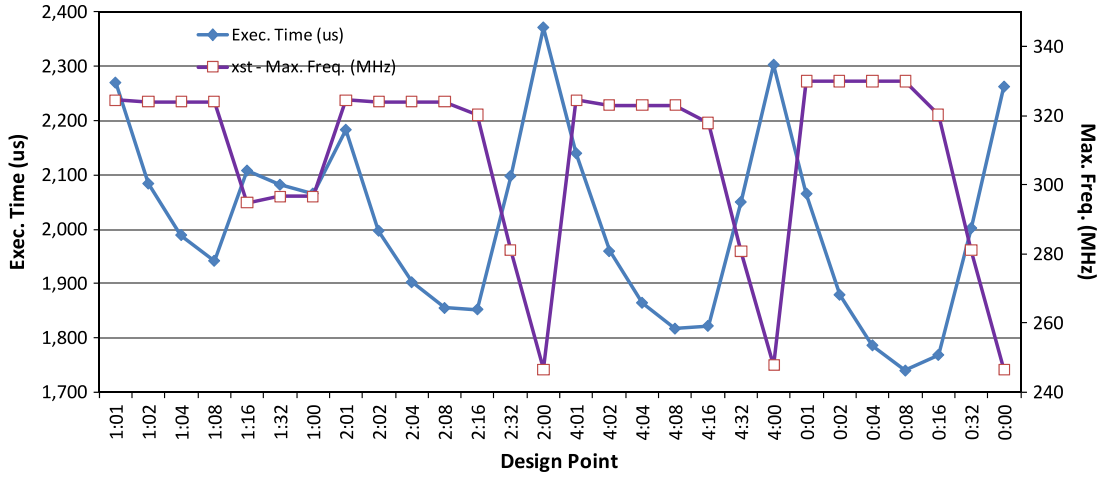

Fig. 20. Experimental results capturing the maximum frequency and the impact on the execution time for the *filter_subband* function considering unrolling of the innermost loops by different unrolling factors.

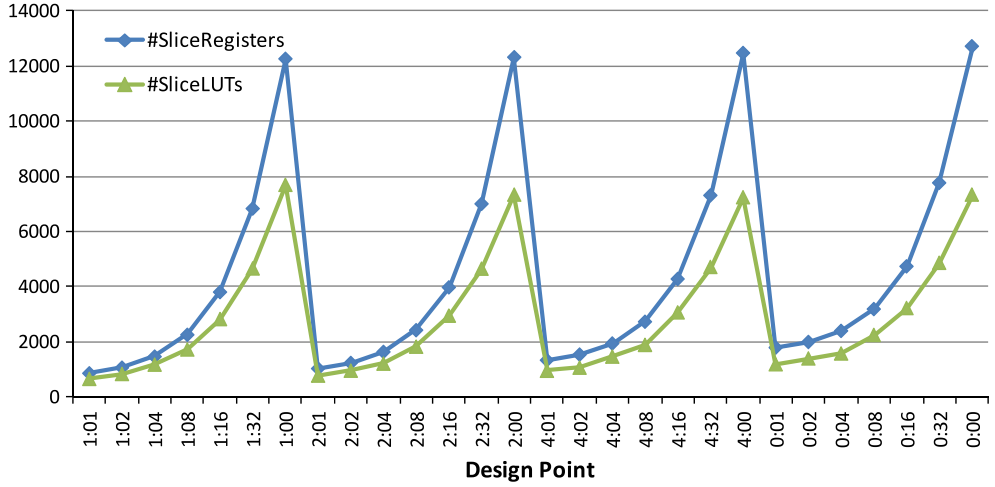

Fig. 21. Experimental results regarding FPGA hardware resources related to LUTs and registers for the designs obtained with strategies considering different loop unrolling factors.

Table 1
Some design points automatically generated and their meaning.

Design point	Innermost loop 1 (rank 1:1)	Innermost loop 2 (rank 2:1)
0:00	Fully unrolled	Fully unrolled
0:01	Fully unrolled	No unrolling
0:n	Fully unrolled	Unrolled $n\times$
k:n	Unrolled $k\times$	Unrolled $n\times$
1:01	Not unrolled	Not unrolled
2:00	Unrolled 2×	Fully unrolled

Table 2
Number of runs for each main toolchain stage and the corresponding global execution time to run the experiments considering the two strategies.

#Design Points (DPs)		28
Common to both Strategies	#Hardware compilation runs	28
	#ModelSim Simulations	28
	Execution Time	~9.66 min.
Strategy considering the execution of all stages for each DP	+ #xst executions	28
	Execution Time	62.79 min
	+ #*P&R* (*map* + *par*) executions	28
	Execution Time	2.44 h
Strategy skipping of some of the back-end stages	+ #xst executions	21
	Execution Time	54.63 min
	+ #*P&R* (*map* + *par*) executions	21
	Execution Time	1.90 h

5. Related work

This section briefly describes the related work concerning compiler optimizations, automated high-level synthesis, and strategies for back-end synthesis, mapping, placement and routing tools.

5.1. Compiler optimizations

Researchers have developed compilation systems for performance tuning most notably in the context of scientific FORTRAN-based programs. While earlier efforts focused on empirical-based approaches (e.g., ATLAS [24]) where programmers would let their applications run and use the output of metrics to decide which sequence of transformations were the best, later efforts focused on more systematic approaches, and used a combination of performance models (e.g., [25]) and special purpose (or domain-specific) languages for defining compiler transformations sequences (e.g., [26]).

Other approaches enable developers to customize the composition and parameterization of design transformations through scripting, in order to automatically derive designs that can meet goals specified by designers [27]. Other projects, (e.g., MULTICUBE [28]) offer a DSE framework for multi-core platforms that can generate multi-objective optimizing strategies based on available performance metrics and constraints. LARA complements the above DSE approaches by providing a unifying DSE platform, which captures and enacts evolving strategies with full design-flow control.

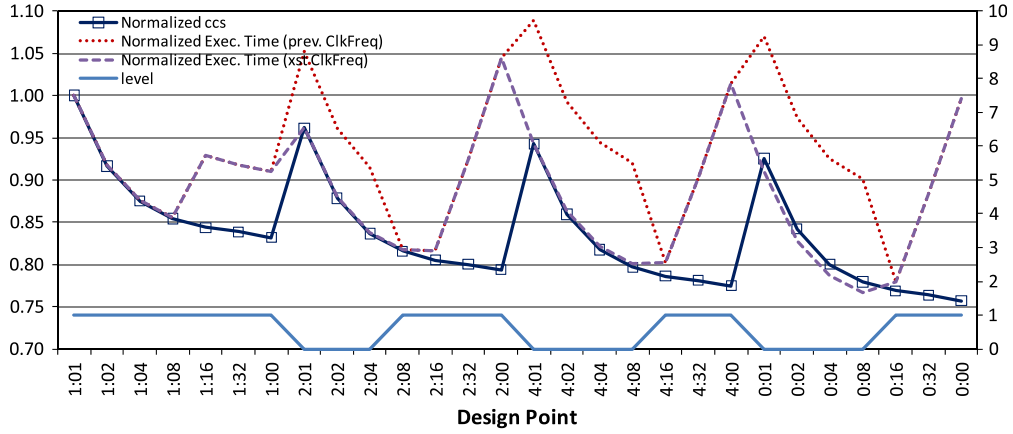

Fig. 22. Experimental normalized results using a DSE strategy that contains a rule to decide about the execution of low level stages (hardware synthesis in this case) of the toolchain.

Table 3
Strategies considered for *filter_subband*.

Design	Strategy
0	Double to float.
1	Double to float + Loopscalar.
2	Double to float + Loopscalar + Full Unroll of first innermost loop
3	Double to float + Loopscalar + Loop unroll (2×) and jam of first nested loops (1 and 1:1)
4	Double to float + Loopscalar + Loop unroll (2×) and jam of both nested loops
5	Double to float + Loopscalar + Loop unroll (2×) and jam of both nested loops + unroll 2 × second innermost loop
6	Double to float + Loopscalar + Loop unroll (2×) and jam of both nested loops + unroll 2 × both innermost loops
7	Double to float + Loopscalar + Loop unroll (2×) and jam of both nested loops + full loop unroll of resultant jammed loop in the first nested loops
8	Double to float + Loopscalar + Double to float + Loopscalar + Loop unroll (2×) and jam of first nested loops (loops 1 and 1:1) + unroll 2 × second innermost loop (2:1)
9	Double to float + Loopscalar + Loop unroll (2×) and jam of first set of nested loops (loops 1 and 1:1) + Loop unroll (4×) and jam of second set of nested loops (loops 2 and 2:1)
10	Double to float + Loopscalar + Loop unroll (2×) and jam of first set of nested loops (loops 1 and 1:1) + Loop unroll (8×) and jam of second set of nested loops (loops 2 and 2:1)
11	Double to float + Loopscalar + Loop unroll (2×) and jam of first set of nested loops (loops 1 and 1:1) + Loop unroll (4×) and jam of second set of nested loops (loops 2 and 2:1) + unroll 2 × second innermost loop (2:1)

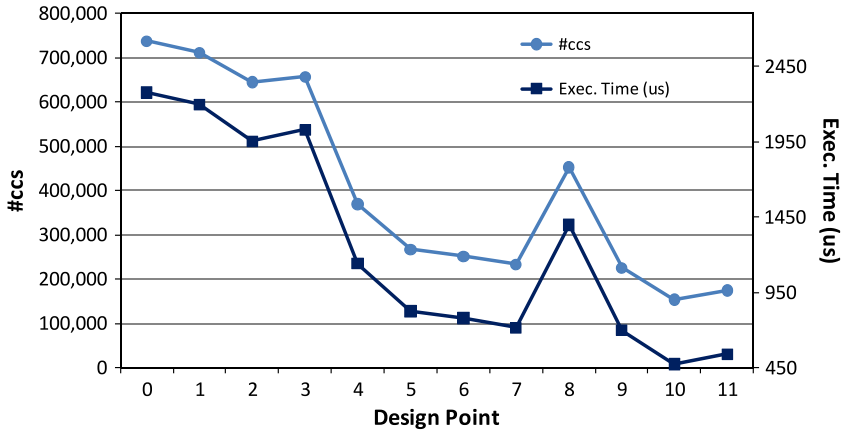

Fig. 23. Experimental results exploring different strategies for the *filter_subband* function and different stages of the design-flow.

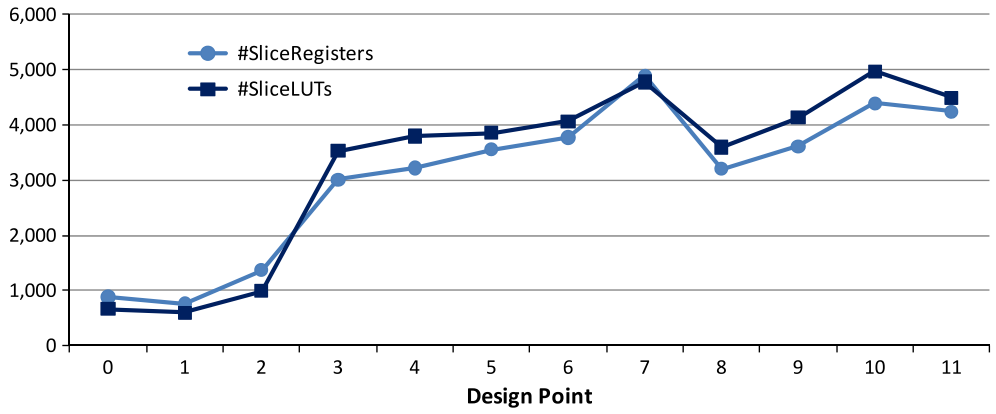

Fig. 24. Experimental results regarding FPGA hardware resources related to LUTs and registers for the designs with strategies mainly based on unroll-and-jam.

5.2. High-level synthesis

Compiling high-level programming languages to FPGAs is a topic extensively addressed by academia and industry (see, e.g., [1] for a survey of representative approaches). Given the large gap between software and hardware abstractions, compilers for FPGAs cannot in general generate efficient customized architectures for complex applications. In addition, the generated hardware depends on the non-functional requirements, which are not embedded in the application forcing designers to explore options and to extensively modify the application code.

There are a number of commercial and open-source tools that synthesize FPGA designs from high-level descriptions. Tools such as Mitrion-C [29], Handel-C [30], and ImpulseC [31] support a subset of C and provide a number of hardware-specific language extensions, to support concurrency, timing and flexible word-length. Libraries (e.g., SystemC [32]) are also used to extend languages to support hardware synthesis.

A number of high-level synthesis tools, such as Trident [33], ROCCC [34], and MaxCompiler [35], target specific application domains. In particular, Trident targets floating-point scientific applications, whereas ROCCC and MaxCompiler target applications that can naturally be modeled to streaming architectures and FIFO channels, such as DSP, digital signal processing and finance applications. Tools such as GAUT [36], C2H [37], Catapult C Synthesis [16,38], and Vivado HLS [39] support behavioral ANSI-C programs, allowing input programs to be compiled with minor software code modifications thus facilitating hardware/software partitioning and exploration. There are also tools specialized in compiler techniques to generate efficient hardware for loops, as is the case of ROCCC [34] and its data reuse techniques.

For all these tools, optimizations can be achieved either through source-annotations (such as pragmas instructing the high-level synthesis tool to unroll and pipeline), by revising directly the code, and through scripting (e.g., Tcl). Such schemes can be captured as part of a strategy and explored in the context of DSE using LARA's single weaving mechanism, and then performed by the weavers according to the specific requirements of each tool in the design-flow.

5.3. Scripting approaches to control tools

To the best of our knowledge our approach is the first to consider an integrated view of controlling and guiding all the tools in a toolchain. There have been, however, approaches for controlling specific tools with respect to directives and constraints. Besides GUI possibilities, most solutions use pragma-based approaches, which pollute the code and does not help maintenance and retargeting. Furthermore, these approaches are not portable to different tools. Others use script-based solutions. One example is the use of Tcl (Tool Command Language) scripts by high-level synthesis tools such as Catapult C [16] and Vivado HLS [39]. While Tcl scripts can be used in a number of situations, they neither provide advanced interfaces between the scripts, the tools, and the input application code (e.g., Catapult C accepts script directives that refer specific loops in the code by using C labels), nor provide an integrated view and an integrated report mechanism that can be feedback to LARA aspects.

5.4. Back-end stages for hardware synthesis

Hardware synthesis tools usually employ their own optimizing strategies. One example occurs with a set of options that can be used in Xilinx ISE tools [18,19] to define strategies devoted to design goals such as area or speed. These strategies are based on a selection of design options provided in each of the tools in the ISE toolchain, using command line, configuration files, or a GUI. Our approach is able to control these options and allow users to define their own custom strategies and perform design-space exploration through the LARA outer-loop mechanism and its integrated scheme to feedback report data.

To the best of our knowledge, the REFLECT project is the first to use an aspect-oriented approach to holistically control and guide design-flows, in order to compile C applications to embedded systems implemented using FPGAs. By extending the possible join points to system artifacts, besides the artifacts in the source code, and by applying to both those types of artifacts actions specified in a programming language, we are exposing users to powerful mechanisms to control and guide the design-flow and to program strategies (mostly defining design patterns) that best suit user requirements.

6. Conclusion

This article described how LARA, a novel aspect-oriented domain-specific programming language, enables a separation of concerns, including non-functional requirements and strategies, for mapping high-level source descriptions to high-performance heterogeneous embedded systems. We described how design-flows can be controlled in LARA. An approach based on aspect-oriented programming, as is the case with LARA, allows developers to preserve the form of the source code thereby enhancing design reuse, and promoting developer productivity as well as architecture and performance portability.

LARA is being evaluated using real-life industrial application C codes including from avionics and audio domains. The experimental results provide strong evidence of the usefulness of our aspect-oriented approach in the context of designing applications that target complex heterogeneous embedded systems.

Furthermore, the integrated environment leveraged by the LARA approach provides a sophisticated and advanced framework for programming and executing Design-Space Exploration (DSE) strategies. The aspect-oriented concepts, scripting support for strategies, the meta-attribute approach, modularity support, and the integrated and transparent view to manipulate and acquire both compiler and synthesis optimizations and options are features well suited for DSE.

Ongoing work is especially devoted to designing LARA-based DSE strategies considering both software and hardware optimizations, including hardware/software partitioning schemes. Future work will consider the integration of more optimizations and code transformations in the weaving phases of the toolchain. We also plan to focus on the specification of LARA strategies that automatically explore code optimizations and transformations for design-space exploration, including the necessary toolchain support.

Acknowledgments

This work was partially supported by the European Community's Framework Programme 7 (FP7) under Contract No. 248976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the European Community. We are grateful to the members of the REFLECT project for their support. Tiago Carvalho and Ricardo Nobre would like to acknowledge the support given by the Portuguese National Science Foundation (FCT) through Grants SFRH/BD/90507/2012 and SFRH/BD/82606/2011, respectively.

References

[1] J.M.P. Cardoso, P.C. Diniz, M. Weinhardt, Compiling for reconfigurable computing: a survey, ACM Comput. Surveys (CSUR) 42 (4) (2010) 1–65 (Article 13).
[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect oriented programming, in: Proc. European Conf. on Object-Oriented Programming (ECOOP'97), vol. 1241, Springer-Verlag LNCS, Finland, June 1997, pp. 220–242.
[3] G. Kiczales, Aspect-oriented programming, in: ACM Computing Surveys (CSUR), Special Issue: Position Statements on Strategic Directions in Computing Research, 1996, 28(4es).
[4] J. Cardoso, et al., LARA: an aspect-oriented programming language for embedded systems, in: Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD'12), Potsdam, Germany, March 25–30, 2012, pp. 179–190.
[5] Z. Petrov, K. Krátký, J. Cardoso, P. Diniz, Programming safety requirements in the REFLECT design flow, in: IEEE 9th Int'l Conf. on Industrial Informatics (INDIN'2011), Portugal, July 2011, pp. 26–29.
[6] J. Cardoso, J. Teixeira, J. Alves, R. Nobre, P. Diniz, J. Coutinho, W. Luk, Specifying compiler strategies for FPGA-based systems, in: 20th IEEE Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM'12), Toronto, Ontario Canada, April 29–May 1, 2012, pp. 192–199.
[7] REFLECT, 2011. <http://www.reflect-project.eu>.
[8] J. Cardoso, et al., REFLECT: Rendering FPGAs to multi-core embedded computing, in: Reconfigurable Computing: From FPGAs to Hardware/Software Codesign, Springer, August 2011, pp. 261–289. (Chapter 11).
[9] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, W. Wong, A high-level compilation toolchain for heterogeneous systems, in: Proc. IEEE Int'l SOC Conf. (SOCC'09), September 2009, pp. 9–18.
[10] ACE CoSy® Compiler Development System. <http://www.ace.nl/compiler/cosy.html>.
[11] R. Nane, V.M. Sima, B. Olivier, R. Meeuws, Y. Yankova, K.L.M. Bertels, DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler, in: Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'2012), Oslo, Norway, August 2012, pp. 619–622.
[12] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, S. Vassiliadis, DWARV: delft workbench automated reconfigurable VHDL generator, in: Proc. 17th Int'l Conf. on Field Programmable Logic and Applications (FPL'07), Amsterdam, The Netherlands, 27–29 August 2007, pp. 697–701.
[13] ACE – Associated Compiler Experts bv., CoSy CCMIR Primer, Ref. CoSy-8134-mirprimer, 2008.
[14] ACE – Associated Compiler Experts bv., CoSy CCMIR Definition, Ref. CoSy-8002-ccmir, 2008.
[15] S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K. Bertels, G. Kuzmanov, E.M. Panainte, The MOLEN polymorphic processor, IEEE Transactions on Computers (November) (2004) 1363–1375.
[16] Mentor Graphics, Catapult C synthesis, 2008. <http://www.mentor.com>.
[17] Rhino, Mozilla Developer Network and Individual Contributors. <https://developer.mozilla.org/en-US/docs/Rhino>.
[18] Xilinx Inc., Command Line Tools User Guide, UG628 (v 14.2) July 25, 2012.
[19] Xilinx Inc., XST User Guide, UG627 (v 11.3), September 16, 2009.
[20] Xilinx Inc., XPower Estimator User Guide, UG440 (v2012.2/14.2) July 25, 2012.
[21] Mentor Graphics, Precision® Synthesis Style Guide, Release 2010a Update1, July 2010. <http://www.mentor.com/>.
[22] T.J. Todman, W. Luk, Reconfigurable design automation by high-level exploration, in: 23rd IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'2012), Delft, The Netherlands, 2012, pp. 185–188.
[23] J.M.P. Cardoso, T. Carvalho, J.G.F. Coutinho, P. Diniz, Z. Petrov, W. Luk, Controlling hardware synthesis with aspects, in: 15th Euromicro Conference on Digital System Design: Architectures, Methods & Tools (DSD'12), Izmir-Turkey, September 5–8, 2012, pp. 226–233.
[24] Automatically Tuned Linear Algebra Software (ATLAS). <http://math-atlas.sourceforge.net/>.
[25] A. Tiwari, C. Chen, J, Chame, M. Hall, J. Hollingsworth, A scalable auto-tuning framework for compiler optimizations, in: Proc. Int'l Symp. on Parallel and Distributed Processing (IPDPS'09), IEEE Computer Society, Washington, DC, USA, 2009, pp. 1–12.
[26] J. Xiong, J. Johnson, R. Johnson, D. Padua, SPL: a language and compiler for DSP algorithms, in: Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'01), ACM, New York, NY, USA, June 2001, pp. 298–308.
[27] Q. Liu, T. Todman, J. Coutinho, W. Luk, G. Contantinides, Optimising designs by combining model-based and pattern-based transformations, in: Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL'09), Prague, Czech Republic, August 31–September 2, 2009, pp. 308–313.
[28] C. Silvano, W. Fornaciari, G. Palermo, V. Zaccaria, F. Castro, M. Martinez, R. Zafalon, S.Bocchio, M. Wouters, G. Vanmeerbeeck, P. Avasare, C. Couvreur, L. Onesti, C. Kavka, A. Turco, U. Bondi, G. Mariani, E. Villar, H. Posadas, C. Y. q. Wu, F. Dongrui, Z. Hao, T. Shibin, MULTICUBE: Multi-objective design space

exploration of multi-core architectures, in: IEEE Computer Society Annual Symposium on VLSI (ISVLSI'10), Kefalonia, Greece, July 5–7, 2010, pp. 488–493. <http://www.multicube.eu/>.
[29] Mitrionics, The Mitrion Accelerated Computing Platform, 2012. <http://www.mitrionics.com/>.
[30] Mentor Graphics, Handel-C Language Reference Manual, 2012.
[31] D. Pellerin, S. Thibault, Practical FPGA Programming in C, Prentice Hall Professional Technical Reference, 2005.
[32] IEEE Std 1666–2005 IEEE Standard SystemC Lang. Ref. Man., 2006.
[33] J. Tripp, M. Gokhale, K. Peterson, Trident: from high-level language to hardware circuitry, IEEE Comput. 40 (3) (2007) 28–37.
[34] G. Villarreal, A. Park, W. Najjar, R. Halstead, Designing Modular Hardware Accelerators in C with ROC-CC 2-0, in: Proc. IEEE Symp on FPGAs for Custom Computing Machines (FC-CM'10), IEEE Computer Society, Washington, DC, USA, April 2010, pp. 127–134.
[35] Maxeler Tech., Maxcompiler White Paper, 2011. <http://maxeler.com>.
[36] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, E. Martin, GAUT: a high-level synthesis tool for DSP applications, in: P. Coussy, A. Morawiec (Eds.), High-Level Syn-thesis: From Algorithm to Digital Circuit, Springer, 2008 (Chapter 9).
[37] Altera Corporation, Nios II C2H Compiler User Guide, November 2009.
[38] T. Bollaert, Catapult synthesis: a practical introduction to interactive C synthesis, in: P. Coussy, A. Morawiec (Eds.), High-Level Synthesis: From Algorithm to Digital Circuit, Springer, 2008 (Chapter 3).
[39] Xilinx Inc., Vivado Design Suite, User Guide, High-Level Synthesis, UG902 (v2012.2) July 25, 2012.

João M. P. Cardoso received his D.Eng. degree from the University of Aveiro, Portugal, in 1993, and his M.Sc. and Ph.D. degrees in Electrical and Computer Engineering from the Technical University in Lisbon (IST/UTL), Portugal in 1997 and 2001, respectively. He is Associate Professor at the Department of Informatics Engineering, Faculty of Engineering of the University of Porto. Before, he was with the IST/UTL (2006–2008), a senior researcher at INESC-ID (2001–2009), and with the University of Algarve (1993–2006). In 2001/2002, he worked for PACT XPP Technologies, Inc., Munich, Germany. He serves as a Program Committee member for various international conferences. He has (co-)authored over 90 scientific publications (including books, journal/conference papers and patents) on subjects related to compilers, embedded systems, and reconfigurable computing. He is currently one of the scientific coordinators of the EU-funded REFLECT research project. He is a member of IEEE, IEEE Computer Society and a senior member of ACM. His research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures.

Tiago Carvalho received an M.Sc. degree in Informatics Engineering from the University of Porto in 2011. He is a Ph.D. student in Informatics Engineering in the same university since 2012. He participated in the AMADEUS project (a research project funded by FCT) and he currently participates in the REFLECT project (FP7 EU funded project). His research interests include programming languages, aspect-oriented programming, and compiler and interpreter technologies.

José G.F. Coutinho received his M.Eng. degree in Computer Engineering in Instituto Superior Tecnico, Portugal in 1997. In 2000 and 2007 he received his M.Sc. and Ph.D. in Computing Science from Imperial College London respectively. Since 2005, Dr. Figueiredo has been working as a research associate at the Custom Computing Research Group in Imperial College London, and has been involved in several UK and EU research projects, including Ubisense, FP6 hArtes, FP7 REFLECT, SmartFlow and FP7 HARNESS. In addition, he has published over 30 research papers in peer-referred journals and international conferences and has contributed as an author to two book chapters, and as an editor of one book.

Ricardo Nobre received an M.Sc. degree in Informatics and Computer Engineering from the *Instituto Superior Técnico* (Technical University of Lisbon) in 2011. He is a Ph.D. student in Informatics Engineering in the University of Porto since 2012. He participated in the VECTOR and AMADEUS projects (research projects funded by FCT) and he currently participates in the REFLECT project (FP7 EU funded project). His research interests include compilation techniques, code transformations and optimizations, and design-space exploration.

Razvan Nane is a Ph.D. Candidate at Delft University of Technology. He is mainly interested in Hardware/Software co-design and High Level Synthesis tools. He is one of the main developers of the DWARV2.0 C-to-VHDL hardware compiler. He received his master degree in Computer Engineering at the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, the Netherlands in 2007.

Pedro C. Diniz received his M.S. in Electrical and Computer Engineering from the Technical University in Lisbon, Portugal and his Ph.D. from the University of California, Santa Barbara in Computer Science in 1997. From 1997 until 2007 he was a researcher with the University of Southern California's Information Sciences Institute (USC/ISI) and an Assistant Professor of Computer Science at the University of Southern California in Los Angeles, California. At USC/ISI was the technical lead of DARPA-funded and DoE-funded research projects, in particular in the DEFACTO project. The DEFACTO project combined the strengths of traditional compilation approaches with commercially available EDA synthesis tools and lead to the development of a prototype compiler for the automatically mapping of image processing algorithms written in programming languages such as C to Field-Programmable-Gate-Array-based computing architectures. He has authored or co-authored many internationally recognized scientific journal papers and over 50 international conference papers. He is currently one of the scientific coordinators of the EU-funded REFLECT research project addressing issues of programmability of reconfigurable architectures.

Zlatko Petrov is a Staff Scientist in Honeywell Advanced Technology Europe with a major focus on avionics platforms, displays and graphics and space technologies. Among the projects in which he is actively taking participation are parMERASA and REFLECT (both EU co-funded). He is project coordinator of the REFLECT project. He previously participated in Merasa (EU co-funded) – time predictable multi-core platform and RECOMP (ARTEMIS JU co-funded) – multi-core systems aiming primarily at reduction of certification and re-certification costs of multi-core based systems, where he was a Principal Investigator from Honeywell's side. His current areas of interest include multi-core and reconfigurable architectures, RTOS, tooling for acceleration of aerospace development cycles, WCET estimation, OLED and flexible displays, multi-modal control of flight displays.

Wayne Luk received his M.A., M.Sc. and D.Phil. in Engineering and Computing Science from Oxford University. Currently Professor of Computer Engineering at Imperial College, he founded and leads the Computer Systems Section and the Custom Computing Group, and is also Visiting Professor at Stanford University. He is on the Program Committee of many international conferences such as FCCM, FPGA and DATE. He has been an author or editor for 6 books, 4 special journal issues, a patent and over 120 research papers in peer reviewed journals and conference proceedings.

Koen Bertels Koen Bertels is head of the Computer Engineering Laboratory of Delft University of Technology in the Netherlands. His research interests focus on heterogeneous multicore architectures and how to use such architectures through a hardware/software co-design process for both embedded and high performance computing application domains.