Controlling Hardware Synthesis with Aspects

João M.P. Cardoso, Tiago Carvalho, José G. Coutinho, Pedro C. Diniz, Zlatko Petrov, and Wayne Luk

Abstract—The synthesis and mapping of applications to configurable embedded systems is a notoriously hard process. Tools 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 understand the interfaces to the various tools and apply, often manually, a sequence of tool-specific transformations making this an extremely cumbersome and error-prone process. This paper describes the use of aspect-oriented techniques for capturing synthesis strategies for tuning the performance of applications’ kernels. We illustrate the use of this approach when designing application-specific architectures generated by a high-level synthesis tool. The results highlight the impact of the various strategies when targeting custom hardware and expose the difficulties in devising these strategies.

Index Terms—Design-Space Exploration (DSE); Automatic Hardware Synthesis; Hardware/Software Partitioning; Program Transformations; Aspect-Oriented Programming; FPGAs.

I. 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 required tools and computing paradigms these architectures expose. The best design solution for inputs with specific data rate characteristics may simply not be feasible for another set of inputs. Non-functional requirements (NFRs), such as reliability, safety, performance, energy consumption, are commonly out of the scope for existing tools or cannot be expressed using current high-level programming languages.

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 by 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.

We have addressed these issues by proposing a novel synthesis toolchain for embedded systems which uses a methodology based on aspect-oriented programming (AOP) [1][2] 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. The input application code is augmented with an aspect-based language specification (the LARA language [3]). 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.

The LARA aspect-based language [3] 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 [4] that are inaccessible in 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 [5].

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. This case study example highlights the flexibility and tool interoperability in our aspect-based framework.

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 follow. In the next section we outline the aspect-based design toolchain by providing a series of examples. These examples illustrate the structure and potential application of aspects in Design-Space Exploration (DSE). Section III describes a case study of the application of DSE to the mapping of a non-trivial kernel code to an FPGA device. In section IV we present experimental results of the use of LARA in the exploration of many hardware designs for this case study. In section V we survey re-
lated work and then conclude in section VI.

II. ASPECT-BASED DESIGN TOOLCHAIN

The REFLECT research project [6][7] aims at supporting the multiple stages of mapping applications described in high-level programming languages such as C\textsuperscript{1} to multi-core embedded architectures. In particular, the project focuses not only on the complete automation of the entire mapping process by building and evaluating the compiler/software toolchain but also at 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 to the key programming concepts we have introduced for hardware synthesis using a domain-specific language, LARA, that relies extensively on aspect-oriented mechanisms.

A. Design Flow in REFLECT

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

- **Input Application**: A functional and executable description of the application. In the current implementation we only support C language sources\textsuperscript{1};

- **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 to satisfy these requirements.

The output of the toolchain is a complete hardware/software design. In the current implementation the software component of the design is also specified in C and it compiles 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 subsequent synthesis tools such as Xilinx’s ISE which ultimately produces a bit-stream that can then be loaded onto an FPGA device for execution. In the context of this paper we use the Catapult-C hardware compiler [8] that takes a stylized C input source code and generates a Verilog design specification.

This design-flow is structured as three major components:

- **LARA Front-End**: The front-end converts LARA descriptions into Aspect-IR (Aspect Intermediate Representation) to be processed by the weavers. The Aspect-IR is a low-level representation in XML format, where 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 [10], 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, and 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, with the latter two common to both software and hardware partitions, which are target architecture independent. The back-end includes assembly code generators from software sections for the general-purpose processor (GPP), and VHDL/Verilog generators for specific hardware cores.

![Figure 1. The REFLECT project design-flow.](image)

A key element of this toolchain is the set of tool-specific weavers that fold the information described in the aspects into a sequence of actions for each individual tool. Weavers can be viewed as control engines that use the information in the aspect to orchestrate the operation of corresponding tools.

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.

B. LARA Aspects: 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 of which aspects to apply and in what order. Ultimately, strategies can be seen as rules that implement specific hardware/software design patterns.

---

\textsuperscript{1}Our approach is also applicable to other imperative programming languages as input code such as MATLAB [9].
Figure 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 includes optimization rules, user knowledge, mapping strategies, target architecture properties, and other information specific to the function. The aspect also specifies two constraints related to input data ranges and noise power.

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

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

A second aspect, presented in Figure 3, instructs the toolchain to fully unroll all innermost for-loops in which the number of iterations is known at compile-time and does not exceed 32. Intuitively, this aspect is generic and can be reused in 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.

```plaintext
aspectdef strategy1
input functionName end
select function[getName==functionName].loop[type=="for"] end
apply optimize("loopunroll","full"); end
condition $loop.numIterIsConstant &&
$loop.num_iter <= 32 && $loop.is_innermost
end
end
```

Figure 3. Aspect module for fully unrolling innermost loops with a number of iterations not or equal to 32.

These two examples highlight the structure of an aspect definition. In addition to the names and possible parameters (the input section), an aspect includes three (3) main sections. A first section, select section, defines the points (or artifacts) in 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 the aspect can manipulate. As an example, a loop can have as 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.

C. Design-Space Exploration in LARA

By extending the portfolio of executable commands, the apply section of a LARA aspect can support the definition and implementation of DSE engines. As illustrated in Figure 4, developers can explore different mapping parameters while exercising the execution of specific tools.

```plaintext
aspectdef ExploreLoopPipelining
input functionName = "filter_subband" end
var ii_to_eval = [1, 2, 3, 4, 5, 8, 12, 16];
select function[getName==functionName].loop end
apply
var exec_time = MAX;
var ii;
for(ii=0; ii<ii_to_eval.length; i++) {
   optimize(opt: "looppipelining", II: ii_to_eval[ii]);
   run("hls-tool", function_name);
   cost_new = report("latency", "clock_freq");
   new_exec_time = cost_new.latency / cost_new.clock_freq;
   if(new_exec_time < exec_time) {
      exec_time = new_exec_time;
      ii = ii_to_eval[ii];
   }
}
println("Selected initiation interval (II): "+ii);
condition $loop.is_innermost end
end
```

Figure 4. Example of a LARA aspect exploring designs resulting from distinct value of pipelining initiation interval.

In this example, the aspect defines a search algorithm for the most effective loop pipelining initiation interval. 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 optimize command) followed by the invocation of a high-level synthesis tool (via the run command). It then observes the resulting design metrics, “latency” and “clock_freq”, by invoking the report command. The best hardware design is then selected using a specific metric of efficiency.

III. 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 the filter_subband function, a hotspot of the MPEG Audio Encoder (MPEG-2 Layers I and II) application. Figure 5(a) 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 subbands. The C code is structured as four 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 possibilities opportunities at the loop level, namely:

- Given constant loop bounds, a compiler can apply strength-reduction to array index calculations and use small bit-width representations for loop control variables;
- Unrolling and the corresponding space of values for the unroll amount for each of the loops. This will expose vast amounts of instruction-level parallelism (ILP) leveraged by concurrent hardware adder and multiplier blocks.

- Privatize the storage associated with the y array in local RAM modules. The values of this array are written in the first loop nest (lines 5-9) and read in the section loop nest (lines 10-14). This transformation will reduce the number of load and store operations by replacing them with internal memory read and write operations.

- Replace the accumulation variables such as the accumulations of values in the y and s arrays by scalar variables (to be promoted to registers).

- For each loop, e.g. in both loop nests and for both inner and outer loops, one can apply hardware pipelining execution as offered as one of the implementation choices of the HLS tool at hand.

For each loop, e.g., in both loop nests and for both inner and outer loops, one can apply hardware pipelining execution as offered as one of the implementation choices of the HLS tool at hand.

```c
void filter_subband(double z[512], double s[32]) {
    double y[64];
    double m[32][64] = {...}; // array of 2,048 constants
    int i, j;
    for (i=0;i<64;i++) {
        y[i] = 0;
        for (j=0;j<64;j++)
            y[i] += z[i+64*j];
    }
    for (i=0;i<32;i++) {
        for (j=0; j<8;j++)
        y[i] = 0.0; y[i] += z[i+64*j];
    }
    for (i=0; i<32; i++)
        for (j=0, j<64; j+=2) {
            y[i] += m[i][j] * y[j];
        }
    for (i=0; i<64; i+=2) {
        for (j=0, j<8; j++)
            y[i] += m[i][j] * y[j];
    }
}
```

Figure 5. (a) Code of the `filter_subband` function from the MPEG encoder application. (b) Use of annotations (in the form of pragmas) to identify specific loops.

Figure 6 depicts the code that results from the transformations specified in the two aspects in Figure 7. These aspects include two options (A and B) for identifying the two loops to be merged (fused). Option A explicitly identifies the loops by using the `position` attribute. Option B identifies the first loop using the `position` attribute and considers that fusion is performed considering both loops in the same hierarchical level.

The aspect presented in Figure 7 starts by defining a variable `functionName` with the name `filter_subband` (line 2). In line 4, 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 Figure 5. For all these loops, two optimizations are invoked: `loopanalysis` and `loopscalar` (see apply section in line 5). The apply section in line 6 also specifies actions to be applied to the same loops given by the select section in line 4. However, in this latter case a condition section (line 7) 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 position 1 (loop in line 5 of Figure 5) will be affected. The other loop to be unrolled twice is the loop correspondent to position “2.1” (see lines 9 and 10). Line 15 has an apply section which invokes the aspect `loopjam` passing as arguments the value of `functionName` and “1.1” and “1.2” (the actual positions of the loops to be fused).

The two options of the aspect `loopjam` are represented in lines 15-22 and 25-30. The first option (Option A) uses the positions of the loops (“1.1” and “1.2”) in the file to identify the loops. The second option (Option B) only uses the position on the original loop nest (i.e., loop in line 7 of Figure 5) 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 5 of Figure 5).

Instead of this indexing scheme with the `position` attribute to identify loops in the code, one can use annotations placed in the original C code. Figure 5(b) shows the `filter_subband` function of Figure 5(a), but with annotations (using pragmas) to identify the three original loops that will be affected by the strategy. Figure 8 shows the strategy defined above, but now with the identification of the loops using the annotations presented in Figure 5(b).

Figure 9 shows a strategy defined with a script that repeats compiler optimizations while changes occur in the actions. The objective is to fully unroll all the for-type loops, with a number of iterations known at compile time that is equal or less than 20, and without inner loops, starting from the innermost loops and iteratively traversing the loop hierarchy.

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 that specific accumulations can be done using fixed precision arithmetic. In addition, they 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 implementation spaces and thus beyond a reasonable manual exploration. In Figure 10 we illustrate how we can extend the code in Figure 5 to support a wide range of strategies 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.

```plaintext
aspectdef filtersubbandStrategy1
    var functionName = "filter_subband";
    select function(name==functionName).loop(type="for") end
    apply optimize("loopanalysis"); optimize("loopscalar"); end
    condition $loop.position == 2.1" end
    apply optimize(opt: "loopunroll", k: 2); end
end
```

Figure 7. Aspects defining a strategy referring loops with the indexing scheme using a position attribute.

```plaintext
aspectdef loopsubbandStrategy1
    var functionName = "filter_subband";
    select function(name==functionName).loop(type="for") end
    apply optimize("loopanalysis"); optimize("loopscalar"); end
    condition $loop.position == 2.1" end
    apply optimize(opt: "loopunroll", k: 2); end
end
```

Figure 8. Aspects defining a strategy for referring to loops by names defined in the annotations.

```plaintext
aspectdef Strategy2
    input functionName="T1" end
    select function(name==functionName) end
    apply do (call repeatloopunroll(functionName)
        while(repeatloopunroll.change); // the use of a default
        // output attribute
        end
    end
end
```

Figure 9. A LARA script performing optimizations while actions produce changes.

```plaintext
aspectdef ExploreStrategies
    input function_name = "filter_subband"
    optimizations = [pipeline, unroll, s1, s2];
    args = [[1],[8],[16]], [[2],[3]], [], [1,2]]
end
select function(function_name) end
apply var exec_time = MAX;
var new_exec_time;
var optID = -1;
/* opts=> 
  [pipeline, [1], [1]], [pipeline, [8]], ..., [s2, [1,2]] */
ext = combine(optimizations , args);
for j=0; i < opts.length; i++) {
    for opt: opts(j)
        if(new_exec_time < exec_time) {
            new_exec_time = cost_new.latency / cost_new.clock_freq
            var optID = j;
            strategyID = i;
            if(strategyID == 2)
                println("Selected strategy id: "+strategyID);
        }
        }
println("Selected strategy id: "+strategyID);
end
```

Figure 10. Example of a LARA aspect for exploring different optimizations and corresponding parameters values.

IV. EXPERIMENTAL RESULTS

This section presents experimental results based on the application of various hardware/software transformations. In particular, we used the aspects described in Sections II and III for the filter_subband kernel code previously described. The hardware designs target a Xilinx Virtex-5 (XC5VFX70) FPGA. The generation of the hardware cores was done using Catapult-C [8]. For the synthesis of RTL-Verilog descriptions to the target FPGA we used the Mentor Graphics Precision 2010a synthesis tool as well as the Xilinx ISE 12.3 placement and routing tools.
The hardware designs are organized in three sets as depicted in Figure 11 (left to right). The first set of designs corresponds to the applications of loop-based code transformations, namely, unrolling, scalar replacement, jam and combinations thereof. Although not exclusively to hardware synthesis, these transformations are traditionally viewed as well suited for hardware synthesis. In a second set of designs, we use aspect strategies that explore hardware-pipelined implementations of the transformed codes by exploring various pipeline initiation intervals. The designs in this set use a uniform exploration of the values of the initiation intervals in the sense that for all loops the same value (whenever feasible) is used in the pipelining. Lastly, for the third set of designs, we expand the search for a non-uniform strategy in which different loops may use distinct value for their pipelining initiation interval.

The results in Figure 11 focus on three performance issues. First, we compare the raw performance of each design against the baseline design (named original) in each set as indicated in the blue bars. Next, we compare the performance over the baseline design thus not subject to any transformation (the baseline is thus the original design of the first set). A second set of results focus on the FPGA resources used as depicted in the plots in the middle section of Figure 11, whereas the bottom section depicts design clock rates and execution latency for the complete hardware design.

As expected, loop-based transformations expose vast amounts of instruction-level parallelism (ILP) leading to designs with substantially enhanced performance and correspondingly use of additional resources when compared to the original base designs (as reflected in the results for the first set). As to the designs in the second and third sets, we observe an increased performance when compared to the best designs in the first set with an almost negligible increase in hardware resources across all its categories (LUTs, FFs, Slices and DSP blocks). We attribute this behavior to a good mapping and scheduling of the synthesis tools used. Lastly, we further observe an overall improvement of the performance of the third set of designs over the corresponding designs in the second set. Furthermore, the designs in this set reveal less variability in terms of performance and resources when compared with the designs in the second set. This is also not surprising as the designs in the third set explore a wider range of initiation interval options. Again, the overhead in terms of FPGA resources is negligible.

Overall, these results reveal that the best choice of transformations is not entirely obvious. Furthermore, as highlighted by the designs in the second set, there is a non-trivial variability of performance for the designs in this set clearly indicating that constrained searches in this rich design space are bridle. A more robust exploration, as exposed by the designs in the third set, requires users to engage in a wider search for which the need for a structured automation approach seems to be clearly a must.

V. RELATED WORK

This section briefly describes the related work concerning compiler optimizations and automated high-level synthesis.

A. Compiler Optimizations

In other contexts researchers have developed compilation systems for performance tuning most notably in the context of scientific FORTRAN-based programs. While the earlier efforts focused on empirical-based approaches (e.g., ATLAS [11]) where programmers would simple 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., [12]) and special purpose (or domain-specific) languages for defining compiler transformations sequences (e.g., [13]).

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 [14]. Other projects, (e.g., MULTICUBE [15]) offer a DSE framework for multicore 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.

B. Hardware Synthesis

Compiling high-level programming languages to FPGAs is a topic extensively addressed by academia and industry (see, e.g., [16] 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 Mitron-C [17], Handel-C [18], and ImpulseC [19] 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 [20]) are also used to extend languages to support hardware synthesis.

A number of high-level synthesis tools, such as Trident [21], GAUT [23], ROCCC [28], and MaxCompiler [22], target specific application domains. In particular, Trident targets floating-point scientific applications, whereas GAUT, ROCC and MaxCompiler targets 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 [23], C2H [24], Catapult C Synthesis [25], AutoPilot [26] and DWARV [27] support behavioral ANSI-C programs, allowing input programs to be compiled without with minor software code modifications thus facilitating hardware/software partitioning and exploration. There are also tools specialized on compiler techniques to generate efficient hardware for loops, as is the case of ROCCC [28] 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.

To the best of our knowledge, the REFLECT project is the first to use an aspect-oriented approach to holistically control and guide a design flow, 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 the user requirements.

VI. CONCLUSION

This paper described how LARA, a novel aspect-oriented programming language, enables a separation of concerns, including non-functional requirements and strategies, for the mapping of high-level applications to high-performance heterogeneous embedded systems. We described how LARA supports hardware/software partitioning, and retargeting in the context of multiple programming languages and design flows. The current LARA prototype is being evaluated with real-life industrial application C codes from the avionics and audio domains. The experimental results provide strong evidence of its usefulness and significance for the mapping of applications
to heterogeneous embedded architectures.

Our experience indicates that 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.

Ongoing work is especially devoted to the integration of all the optimizations and code transformations in the weaving phases of the toolchain. Future work will focus on the specification and toolchain support for LARA strategies that automatically explore code optimizations and transformations for design space exploration.

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. In particular, we acknowledge the support of Tiago Lobo for his work on hardware synthesis measurements.

REFERENCES


