LARA: An Aspect-Oriented Programming Language for Embedded Systems

João M. P. Cardoso,
Tiago Carvalho
Universidade do Porto,
Faculdade de Engenharia (FEUP),
Dep. de Engenharia Informática
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jmpc@acm.org,
tiago.diogo.carvalho@fe.up.pt

José G. F. Coutinho,
Wayne Luk
Department of Computing,
Imperial College London,
180 Queen's Gate,
London SW7 2BZ,
United Kingdom
gabriel.figueiredo@imperial.ac.uk,
w.luk@imperial.ac.uk

Ricardo Nobre,
Pedro C. Diniz
INESC-ID,
Rua Alves Redol 9
1000-029 Lisboa, Portugal
rjfnobre.disk@gmail.com,
pedro@esda.inesc-id.pt

Zlatko Petrov
Honeywell International s.r.o,
Turanka 100
627 00 Brno
Czech Republic
zlatko.petrov@honeywell.com

Abstract

The development of applications for high-performance embedded systems is typically a long and error-prone process. In addition to the required functions, developers must consider various and often conflicting non-functional application requirements such as performance and energy efficiency. The complexity of this process is exacerbated by the multitude of target architectures and the associated retargetable mapping tools. This paper introduces an Aspect-Oriented Programming (AOP) approach that conveys domain knowledge and non-functional requirements to optimizers and mapping tools. We describe a novel AOP language, LARA, which allows the specification of compilation strategies to enable efficient generation of software code and hardware cores for alternative target architectures. We illustrate the use of LARA for code instrumentation and analysis, and for guiding the application of compiler and hardware synthesis optimizations. An important LARA feature is its capability to deal with different join points, action models, and attributes, and to generate an aspect intermediate representation. We present examples of our aspect-oriented hardware/software design flow for mapping real-life application codes to embedded platforms based on Field Programmable Gate Array (FPGA) technology.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – Frameworks. D.3.3 [**Programming Languages**]: Processors – Compilers, Retargetable Compilers, Optimization, Code Generation. C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems, Microprocessor/microcomputer applications. B.7.1 [**Integrated circuits**]: Types and Design Styles – Algorithms implemented in hardware.

General Terms Design, Experimentation, Languages.

Keywords Aspect-Oriented Programming; Compilers; Reconfigurable Computing; FPGAs; Embedded Systems; Domain-Specific Languages

1. Introduction

The development and mapping of applications to contemporary heterogeneous high-performance embedded systems requires tools with very sophisticated design-flows that are aware of critical applications requirements, both functional and non-functional (e.g., real-time performance and safety) while meeting target architecture's stringent resource constraints (e.g., storage capacity and computing capabilities).

This development and mapping process must consider a myriad of design choices. Typically, developers must partition the application code among the most suited system components – a process commonly known as hardware/software partitioning [1]. Subsequently, they have to deal with multiple compilation tools (sub-chains) that target each specific system component. These problems are exacerbated when dealing with FPGAs (Field-Programmable Gate Arrays), a popular technology which combines the performance of custom hardware with the flexibility of software [2][3]. As a consequence, developers must explore code and mapping transformations specific to each architecture so that the resulting solutions meet the overall requirements. As the complexity of emerging heterogeneous embedded systems continues to grow, the need to meet increasingly challenging design trade-offs (e.g., weight, size, energy efficiency, reliability and performance) will undoubtedly exacerbate the complexity of the mapping of sophisticated applications to these embedded systems.

In practice, this approach leads to code that is transformed beyond recognition and where developers have manually applied an extensive set of architecture-specific transformations and tool-specific directives. However, such practice leads to low developer productivity and, more importantly, limited application portability. For instance, when the underlying architecture changes developers may need to restart the design process.

This paper describes LARA, a novel aspect-oriented programming (AOP) [4] language for mapping applications to heterogeneous high-performance embedded systems. This language allows developers to capture non-functional requirements from applications in a structured way, leveraging high-level abstractions such as hardware/software design templates and flexible toolchain interfaces. Developers can thus benefit from retaining the original application source while exploiting the automation benefits of various domain-specific and target component-specific compilation/synthesis tools. In essence, LARA uses AOP mechanisms to offer in a unified framework (a) a vehicle for conveying application-specific requirements that cannot otherwise be specified in the original programming language for design capture, (b) using these requirements to guide the application of transformations and mapping choices, thus facilitating design-space-exploration (DSE), and (c) interfacing in an extensible fashion the various compilation/synthesis components in the toolchain. This paper makes the following specific contributions:

- Introduces LARA, an AOP language capable of capturing transversal (across multiple codes) and vertical concerns (across different stages of a design flow);
- Describes the LARA join point model that can include not only program execution points but also hardware system components and their properties;
- Extends the join point model with attributes used and defined in the aspects to codify complex strategies;
- Covers a programmable and systematic approach to control and guide different design flow actions and/or application of code transformations;
- Presents experimental results of the use of LARA for a set of real-life codes targeting a heterogeneous embedded architecture with both general-purpose processor (GPP) and FPGA technologies, using commercially available tools for compilation and hardware synthesis.

The results described in this paper reflect the development of the LARA language in the context of the REFLECT (REndering FPGAs to MuLti-Core Embedded CompuTing) research project [5][6][7]. The applications selected in our study and thus the corresponding requirements were drawn from two of this project's industrial partners, targeting the Avionics and Audio domains. Specifically, the LARA AOP approach has been designed to help developers to reach good design solutions with low programming effort. This experience of using aspects for hardware-oriented transformations reveals the benefits of AOP in: (a) application program portability across architectures and tools, and (b) productivity improvement of developers and programmers.

This paper is organized as follows. Section 2 describes a motivating example. Section 3 presents the design flow for our approach. Section 4 describes our AOP approach. Section 5 focuses on the LARA aspect language, providing a number of illustrative examples. Section 6 focuses on the practical impact of our approach. In Section 7, we summarize related work and Section 8 then concludes.

2. Compilation/Synthesis Example

We now present a motivational example that highlights some of the challenges faced by application developers when targeting embedded systems. In this example, we assume that the developer has already carried out a performance analysis study and has selected a set of computational kernels that can benefit from either hardware acceleration or source-level code transformations.

We consider a particular implementation of the MPEG-2 Audio Encoder (layers I and II) in C. One of the hot-spots is the polyphase filter bank function depicted in Figure 1. This function is structured as two doubly nested `for`-loops that manipulate four one- and two-dimensional array variables of 64-bit floating-point values. It processes 512 audio samples and outputs 32 equal-width frequency subbands. The statically allocated *m* array holds the filter coefficients.

```
1. void fsubband(double z[512], double s[32]){
2.  double y[64];
3.  int i,j;
4.  static const double m[32][64] = {...};
5.  for(i=0;i<64;i++) {   // loop1
6.    y[i] = 0;
7.    for(j=0;j<8;j++)    // loop2
8.      y[i] += z[i+64*j];
9.  }
10. for(i=0;i<32;i++) {   // loop3
11.   s[i] = 0;
12.   for(j=0;j<64;j++)   // loop4
13.     s[i] += m[i][j] * y[j];
14. }
15.}
```

Figure 1. C source code of the function *fsubband* in the MPEG-2 audio encoder application.

An implementation of this kernel for an embedded system with a GPP coupled to an FPGA (e.g., acting as a hardware accelerator) could explore two mapping scenarios, namely[1]: (1) the complete mapping of the code to the GPP leveraging software-only GPP compiler optimizations, and (2) the generation of an application-specific architecture implemented on the FPGA derived from partitioning the application between the GPP and the FPGA hardware.

Each of these mapping scenarios may require different strategies in terms of how the computation is partitioned between the target components (GPP vs. FPGA), and consequently how data manipulated by the sub-computations are organized and partitioned. Furthermore, as a multi-computing architecture, once the data and the computation have been partitioned, data communication between the components and their subsequent synchronization need to be considered and included in the mapped code.

Once the partition and mapping have been decided, each of the partitioned computations may still be subject to further transformations. The computations to be executed on the GPP can be subject to a wide variety of transformations offered by C compilers such as *gcc*. These include loop-based transformations (e.g., loop unrolling and/or software pipelining), data type conversions (double to float or double/float to fixed-point), and even array to memory mapping and caching in local scratch-pad memories.

With respect to computations mapped to the FPGA-based hardware accelerator, there is a wider range of compilation and synthesis options that can be exercised which further increase the complexity of this mapping process. For example, in the accumulation statements in lines 8 and 13 in Figure 1, one could cache (scalar replace) the values associated with the *y* array thus substantially reducing the number of load/store operations from/to external memories. In the presence of dual-ported on-chip memories, local arrays *m* and *y* could be mapped to distinct memories. Loop transformations could then be used to expose concurrent accesses to the *m* and *y* arrays. To achieve this mapping result, a developer would define a strategy to combine unroll-and-jam to *loop1* and unrolling to *loop4* followed by a specific mapping of array variables to internal FPGA storage (e.g., Block RAMs - Random Access Memories - in a Xilinx FPGA).

[1] A third scenario where the entire computation is mapped to the FPGA would, in practice, be infeasible.

These mapping strategies and the associated compiler and mapping transformations highlight the interplay between them and the complexity in assessing their potential performance and resource use. To address this complexity we developed the LARA AOP language. LARA allows developers to control the tools in a compilation toolchain and to apply a wide range of transformations and target architecture mapping choices in an automated fashion. We now illustrate how a developer could specify examples of these transformations using LARA.

The first of these aspects, depicted in Figure 2, instructs a weaver (Figure 5, see Section 3) to fully unroll all innermost for-type loops in which the number of iterations is known at compile-time and does not exceed 32. For instance, when this aspect is applied to function *fsubband*, the weaver will fully unroll *loop2* shown in Figure 1. This aspect is generic and can be reused for other functions/applications as part of an optimization strategy. The developer can even increase its potential reuse by defining the number of iterations (32 in this example) and the unrolling factor as two additional aspect input parameters.

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

Figure 2. Aspect module that fully unrolls innermost loops with the number of iterations less than or equal to 32.

The second aspect, shown in Figure 3, focuses on the software/hardware partition problem where functions and code sections can be selected and mapped to hardware specified in a hardware description language such as Verilog or VHDL. This particular aspect also conveys to the compiler and synthesis tools that are part of the design flow (see Section 3), information about the interval range of variable “z”, the maximum noise power allowed and the data type of argument “s”. This information can be used by a word-length analysis engine (here identified as *datarepr*) to derive customized word-lengths by exploring acceptable precision and accuracy values. Lastly, in Figure 4, we present an aspect for the mapping of array variables to local storage for a specific target hardware architecture: an internal Block RAM in a Xilinx Virtex-5 FPGA device.

Regarding code generation, the weaver component included in our compilation infrastructure (see Section 3) creates two source code partitions: a software partition which is compiled by the native target GPP compiler, and a hardware partition which is processed by a target hardware architecture synthesis tool responsible for generating the corresponding bit-level device configuration, able to program the FPGA.

This example highlights a set of features of our combined hardware/software aspect-oriented mapping approach:

- It enables the specification and control of software- and hardware-related transformations with specific parameter values, such as the amount of unrolling.
- It supports the identification of code sections to be mapped to classes of computation nodes, such as traditional GPPs and hardware accelerators, facilitating the interface and interplay of diverse front-end and back-end tools such as source-to-source compilers, code generators, and hardware synthesis tools.
- By maintaining a single source code, the approach allows the composition of transformations and strategies, thus promoting code portability.

```
aspectdef maximizePerformance
 input funcName = “fsubband” end
 select
   function{name==funcName}.arg{name==”s”} end
 apply $arg.noise_power <= 1E-3;
       $arg.def type=”float”; end
 select
   function{name==funcName}.arg{name==”z”} end
 apply $arg.range = “[-40..120]”; end
 select function{name==funcName} end
 apply
   $function.map(to: “hardware”, id: "virtex5");
   call strategy2;
   optimize(“datarepr”);
   call map2BRAMs(funcName);
 end
end
```

Figure 3. Aspect module used to map to hardware and optimize the *fsubband* function using range values.

```
aspectdef map2BRAMs
  input func_name end
  var id=1;
  select function{name==func_name}.var end
  apply
    $var.map(to: “Memory”, type:”BRAM”,
             ports: 2, id: id++);
  end
  condition $var.isarray &&
            $var.scope == “local” &&
            $var.size <= 2048 &&
     ($var.parloads >=2 || $var.parstores >=2)
  end
end
```

Figure 4. An aspect module used to map specific array variables to local on-chip block RAM.

Although not highlighted in this example, our compilation and synthesis approach also enables the developer to engage in design space exploration (DSE). In this specific example, loop-unrolling exposes additional instruction-level parallelism (ILP) opportunities for the hardware-based solution, but may increase the amount of required hardware resources and the associated code as it expands the source program. Similarly, mapping variables to local storage (in effect caching them) may reduce the number of memory accesses at the expense of an increased amount of storage resources. As such, developers can repeatedly use the same aspect with a wide range of parameter values and modify the sequence of application of the aspects in search of a design that meets specific overall requirements.

We next describe the basic structure of the compilation and synthesis design flow we have developed, highlighting the ability of the aspect-oriented approach to control a wide range of transformations and tools.

3. Design Flow

LARA, the AOP language described in this paper, was developed in the context of the compilation and synthesis design flow for the REFLECT research project [6][7]. One of the goals of REFLECT is to map applications described in high-level programming languages such as C[2] to multi-core embedded architectures. This mapping, and subsequent compilation/synthesis, invariably makes use of a wide variety of tools with unique features and interfaces. LARA has been designed to capture non-functional requirements and to guide tools so that developers can quickly achieve design solutions that meet these requirements, which cannot be easily expressed using common programming languages such as C. In addition, LARA allows the definition of strategies specifying which aspects to apply and in what order. Ultimately, strategies can be seen as rules that implement specific design patterns.

We now describe in more detail the overall design flow, highlight its main components and explain how LARA enables their effective use in developing feasible embedded systems design solutions. A detailed description of LARA can be found in Sections 4 and 5.

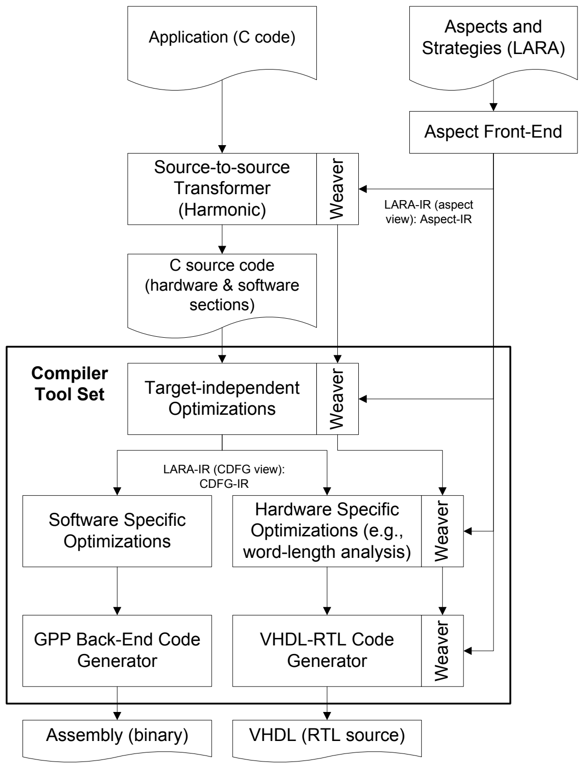
As shown in Figure 5, our design flow accepts two types of source descriptions as inputs: **(1) Input Application**: The current implementation supports C sources (C99 std. compliant); **(2) LARA Description**: The LARA descriptions capture non-functional requirements in the form of aspects and strategies. They enable developers to define application characteristics such as precision representation, input data rates or even reliability requirements for the execution of specific code sections.

This design flow chain is structured as three major components, namely:

- **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 of LARA in XML format, where information is structured in a way to facilitate the parsing of aspects and strategies.
- **Source-to-source transformer**: This stage, using the Harmonic tool (based on [9]), performs 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 (such as remote procedure calls) to enable communication between software and hardware components. The results of this stage are source files reflecting the partitioning, and additional code generated to realize synchronization and communication between 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 CoSy [10] compilation framework is currently being used. The back-end includes assembly code generators for the GPP (software sections) and VHDL/Verilog generators for specific hardware cores.

The design flow includes several weavers at different levels of the toolchain. Each weaver receives as input: C source code or an IR and the Aspect-IR, and outputs: the transformed C code or the transformed IR and if required a modified Aspect-IR for the next weaver in the sequence.

[2] Although our compilation framework is applicable to other imperative programming languages, due to the constraints imposed by the availability of the front-end we are only focusing on the C programming language. We have, nevertheless validated the same approach for MATLAB [8].



Application (C code)
Aspects and Strategies (LARA)
Aspect Front-End
Source-to-source Transformer (Harmonic)
Weaver
LARA-IR (aspect view): Aspect-IR
C source code (hardware & software sections)
Compiler Tool Set
Target-independent Optimizations
Weaver
LARA-IR (CDFG view): CDFG-IR
Software Specific Optimizations
Hardware Specific Optimizations (e.g., word-length analysis)
Weaver
GPP Back-End Code Generator
VHDL-RTL Code Generator
Weaver
Assembly (binary)
VHDL (RTL source)

Figure 5. LARA based Design Flow.

4. The LARA Approach

A key innovation of our aspect-based approach lies in bringing together, in the same framework, various types of transformation and operational aspects in the mapping of computations to embedded systems. Specifically, LARA allows developers to specify the following types of aspects:

- **MONITORING:** Specification of which implementation features, such as current value of a variable or the number of items written to a specific data structure, provide insight for the refinement of other aspects;
- **SPECIALIZING:** Definition of specific properties for a particular input code when targeting a specific system (e.g. specializing data types, numeric precision and input/output data rates);
- **MAPPING AND GUIDING:** Specification of design patterns, which embody mapping actions to guide tools to perform specific implementation decisions (e.g. mapping array variables to memories; using FIFOs to communicate data between cores; leveraging dynamic reconfiguration techniques for performance or using temporal/spatial redundancy for fault-tolerance);
- **RETARGETING:** Specification of characteristics of the target system in order to make the tools adaptable and aware of those characteristics, as well as facilitating implementation on other systems.

LARA relies on the main concept of aspects, generically defined by the following statement:

"In programs P, whenever condition C arises, perform action A." [11]

Associated with AOP are usually the notions of pointcut and advice. A pointcut exposes points of interest (join points) related to a program. Join points refer to points in the code and/or in the program execution. An advice refers to the actions to be performed for each join point exposed by the pointcut mechanism.

In addition, AOP defines the notion of a join point model, which defines the points of interest for a given programming language. A typical join point model includes program constructs and structures such as function calls, fields in a class, and functions. In our approach, we consider a join point model that captures most structures and constructs (e.g., loops, conditionals, variables, array accesses) found in C in order to specify actions that target complex applications containing such code artifacts.

Our approach also considers points of interest as points in the execution of a program as well as in the target physical system. These points may include components of the system, such as a microprocessor, and the system's parameters, such as its specific inputs. As such our AOP approach can be thought as:

"In programs P and/or systems S, whenever condition C arises, perform action A."

The following are a few examples that can be captured by our AOP approach:

- For each variable of type double in function *f1*, change the type to float.
- Set noise power of parameter *s* of function *f1* to 1E-3.
- Set microprocessor clock frequency to 400 MHz.
- Map arrays of functions migrated to hardware, with size < N, to BRAMs (local memories).

Figure 6 illustrates the LARA front-end which converts a LARA aspect file into Aspect-IR. To perform this conversion, the front-end requires three specification files: (1) the join point model representing the points of interest in the input programming language and in the target architecture; (2) the join point attributes defining properties associated with each join point type; and (3) the action model describing each possible action that an aspect can perform on a join point. We describe each of these models next.

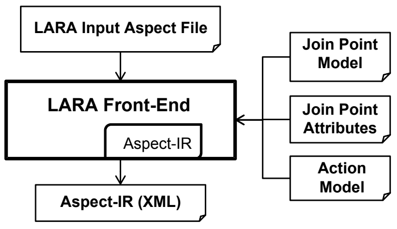


Figure 6. The LARA Front-End component.

4.1 Join Point Model

The points of interest in the program code and/or program execution are specified in the join point model, which is structured hierarchically in an XML file. Since the join point model specification describes join point types and their hierarchy, it can be used to validate pointcut expressions. Rather than hardcoding types in the grammar, this approach allows the model to be easily updated and expanded. Also, by accepting a join point model file externally to the front-end, it is possible to reuse the LARA front-end for other programming languages (see [8]) and with different system components and architectures.

Figure 7(a) shows an excerpt of the join point model currently used for C programs. In this case, the join point type *loop* has as its predecessor the *body* of a *function*, followed by the *file* it belongs to.

```
file
  |\_var
  |\_declaration
   \_function
        |\_prototype
         \_body
            |\_first
            |\_last
            |\_var
            |\_call
            |\_if
            |   |\_condition
            |   |\_then
            |    \_else
             \_loop
                |\_init
                |\_condition
                |\_counter
                |\_body
                 \_control
```

(a)

```
attributes
 |\_var
 |    |\_name
 |    |\_type
 |    |\_is_array
 |    |\_is_pointer
 |    |\_is_in
 |    |\_value
 |     \_is_out
 |\_function
 |    |\_name
 |    |\_num_lines
 |     \_return_type
 |\_call
 |    |\_name
 |    |\_return_type
 |    |\_num_argin
 |     \_num_argout
  \_loop
      |\_type
      |\_num_iterations
      |\_increment_value
       \_nested_level
```

(b)

Figure 7. Excerpt of the input models: (a) join point model definition for C programs; (b) the attribute model.

In addition to the join points related to the C program (as illustrated in Figure 7(a)), our approach also considers join points related to system components. Examples of those join points are the GPP (General Purpose Processor), the CCU (Custom Computing Unit), and the Memory.

4.2 Join Point Attributes

The attributes associated with each join point type are specified in the join point attributes file. Figure 7(b) and Table 1 illustrate examples of attributes for some of the join points supported by our join point model. Attributes are properties in which values are either statically or dynamically known. These values can be used to filter join points (e.g., a `for` type loop is defined as a join point *loop* with attribute *type* with value "for") in conditions that trigger the use of a certain action on the aspect, and also as arguments for the apply sections of an aspect.

Table 1. Artifact list containing examples of join point attributes associated with system components.

Join point	Attributes	Examples of values
GPP (general purpose processor)	name	PowerPC
	family	P440
	clk_freq	400 MHz
CCU (custom computing unit)	id	1
	clk_freq	100 MHz
	max_slices	4,000
Memory	num_ports	1
	max_size	256 × 32-bit
	type	BRAM, DRAM, Distributed RAM

In addition, attributes expose information for each join point, and that information can be obtained by the weavers in the compilation flow and/or can influence the use of actions. For example, the join point type *var* has properties including the *name of the variable*, the *number of readings and writings*, and *the initialized value*. Another example is the information we can obtain about a loop. With the *num-*

ber of iterations it is possible to choose whether an aspect should perform loop unrolling or not. With the nested level attribute it is possible to select specific loops.

There are global attributes that are common for all join points in a program, such as the name of the file or the total number of array references. As with the join point model, the attributes specification file allows the aspect language to be updated and expanded more easily.

4.3 Action Model

The action model specifies all actions that can be applied to join points. For instance, when optimizing code by using the loop unrolling action, the tool should specify which stage and which engine can perform this optimization. In our current design flow we consider six types of actions:

- **INSERT** allows arbitrary code to be instrumented before, around or after a specific join point. This action is used mainly for monitoring-based aspects.
- **DEFINE** allows existing attributes to be modified, and new attributes to be created.
- **MAP** allows developers to associate computations and data structures to specific hardware components.
- **OPTIMIZE** allows a number of compiler transformations and optimizations (e.g., loop unrolling, function inlining/outlining, scalar replacement, loop fusion) to be performed on a specific set of join points.
- **REPORT** instructs the weaver to generate an aspect file with a set of attribute definitions (see above) with the values of attributes for selected join points. This action is particularly useful for sharing data across weavers, and to perform feedbacks in the design flow.
- **CALL** invokes an aspect, allowing input arguments to be passed, and output arguments to be accessed.

The action model is specific to a design flow and makes the LARA front-end aware of what tools and which arguments to use. The action model for each compiler optimization (related to the *optimize* action) defines the name of the compiler engine and the possible parameters. Example parameters are: *loopunroll* and unroll factor; *tiling* and size of the block; *loop-pipelining* and initiation interval (II); target clock frequency and function *inlining*/*outlining*. In summary, by having a join point model, join point attributes, and an action model independent of the LARA front-end, we improve flexibility and adaptability of different programming languages, target systems, and compilation/design flows.

5. LARA Language Description

In this section we provide details about the LARA language. The complete specification of LARA has been included in a report [12].

5.1 Aspect Definition

The current version of the LARA language is compliant with the grammar partially depicted in Figure 8. An aspect file is composed of three sections: the import declarations, the definitions of aspect modules, and code definitions. The import declaration section is optional and allows references to external aspects; the section with aspect definitions, on the other hand, specifies for each aspect the join points to be captured and the actions to be performed on them when certain requirements are fulfilled; finally the code definition section, also optional, includes code to be injected into the source code.

An aspect definition is declared using the *aspectdef* keyword. Here, developers can define pointcut expressions and also the actions to take place on the selected join points. In the current version of the LARA language, an aspect definition can have *Select*, *Apply* and *Condition* sections. These sections have dependences between them. For instance, the *Apply* section can be associated with one or more pointcut expressions (*Selects*). Also, a *Condition* section is used to validate an *Apply*, i.e., to enable or disable an action over a join point. In general, we can have various applies to the same select, and an *Apply* associated with more than one select. Not all applies need to have a condition section.

Each aspect definition can start with the declaration of input and output parameters. These parameters are used to pass values to aspects and to program aspects to return values, respectively. Each aspect has four additional optional sections for: declaring variables, declaring functions, code to initiate (prolog) the execution (*initialize*) and for code to terminate (epilogue) the execution (*finalize*) of the aspect.

```
Start    = {Import}, AspDef, {AspDef}, {Code-
           Def};
Import   = ’import’, Identifier, {’.’, Iden-
           tifier} ‘;’ ;
AspDef   = ‘aspectdef’, Identifier, Input,
           Output, {VarDecl}, {FunctionDecl},
           Initialize, AspBody, {AspBody}, Fi-
           nalize, ‘end’
AspBody  = Select, {Select}, Apply, {Apply},
           {ConditionExpr} ;
CodeDef  = ‘codedef’, Code, ‘end’;
```

Figure 8. Excerpt of the LARA language grammar.

Figure 9 depicts an aspect definition that inserts a *printf()* statement immediately before each function call. To capture the intended join points a *Select* statement is used. The *Select* statement is identified by a label (in our example, `allFunctionCalls`) and referred by *Apply* statements, and includes a pointcut expression that defines the join points to be captured. The pointcut expression is validated by the front-end using the join point model specification (Section 3).

Developers do not need to specify in LARA the complete join point hierarchy in the pointcut expression, as the front-end auto-completes the entire hierarchy using the join point model specification as a reference. Also, the pointcut expression can use, for a specific selection, join point attributes to filter the selection according to specific attribute values. For instance, the pointcut expression in Figure 9(a) uses the attribute *name* to specify all function calls. As shown in this example, the code section (between tags *%{* and *}%*) allows the use of parameters (between *[[* and *]]*) that will be replaced by the weaver with the corresponding values.

The corresponding Aspect-IR code is shown in Figure 9(b). LARA has been designed for code compactness, legibility and flexibility. Aspect-IR, on the other hand, was designed to facilitate the parsing and processing of aspect definitions by weavers. As explained before, the LARA front-end automatically generates the corresponding Aspect-IR from a LARA description that will be subsequently passed to the weavers in the design flow.

One type of action associated with a pointcut expression is the insertion of C code into the source code (Figure 9*(a)*) with the option to use values of join point attributes. This insertion can be done *before*, *after* or *around* the join point. Target code can be placed between brackets after the *insert* command, as depicted in Figure 9*(a)*, or in a separated structure called *codedef*. The LARA front-end does not take actions over the code by itself and therefore the insertion action is passed to the Harmonic weaver.

```
aspectdef monitoringCall
 allFunctionCalls: select function.call end
 applyAllCalls: apply to allFunctionCalls
  insert before
   %{printf("call to [[$call.name]]\n");}%; end
end
```

(a) LARA description

```
<aspects>
 <aspect name="monitoringCall">
   <pointcut name="allFunctionCalls"><file>
     <function>
       <body> <call name=".*"/> </body>
     </function></file></pointcut>
   <apply to="allFunctionCalls"><actions>
     <insert
      code="monitoringCall_applyAllCalls_0.txt"
      stage="harmonic" position="before">
        <parameter target="function"
          attribute ="name"
          replace="<$function.name>"/>
     </insert></actions></apply>
</aspect></aspects>
```

(b) Aspect-IR description

Figure 9. (a) Simple LARA code that performs instrumentation; (b) corresponding Aspect-IR.

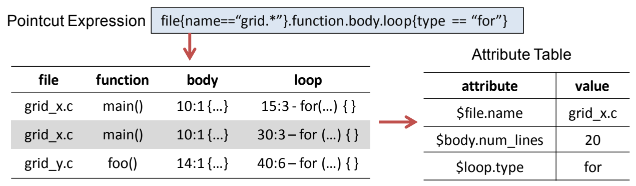
Another supported action is *optimize* (Figure 2). This feature specifies compiler source-level transformations and optimizations, such as *loop unrolling* and *function inlining*. The *loopunroll* action triggers the unrolling of selected loops using a factor number specified by the developer. The *inline* option inserts the body of the function to inline on the location of the function call.

Besides the assignments to existing join point attributes without directing affect the representation of a program, our current action model also includes the *define* action (*def* keyword used in Figure 3). This leads, whenever possible, to modifications of the program representation (e.g., defining the type of a variable as in the aspect in Figure 3). Assigning values to join point attributes also provides a way to share information between different stages of the design flow. For example, specific stages can set values of certain attributes so that they can be used by subsequent stages of the design flow.

To limit *Apply* statements, developers can define conditions. Conditions can be regarded as logical expressions that define the triggering conditions for actions specified in an *Apply*. Join point attributes can also be used in conditions. Examples of actions dependent on specific conditions are: (a) apply *loop unrolling* only if the loop is of type "for" and has at most 8 iterations; (b) insert code before a function call if it returns a float; (c) *inline* a function if it does not contain loops. Additional examples of the use of conditions are illustrated in Figure 2 and Figure 4.

5.2 Join Point Chains

One novel feature of our approach is that pointcut expressions are written in a form that reflects the hierarchy defined in the join point model (see Section 4.1). For instance, in Figure 10 the pointcut expression has 4 elements connected in a chain: the **file** which captures all files with names starting with *grid*, the **function** which includes all functions enclosed in those files, the **body** which reflects these functions definitions, and the **loop** which captures `for`-type loops in those functions. This hierarchical mechanism has similarities to the *within*/*within code* used in AspectJ.



file	function	body	loop
grid_x.c	main()	10:1 {...}	15:3 - for(...) { }
grid_x.c	main()	10:1 {...}	30:3 – for (...) { }
grid_y.c	foo()	14:1 {...}	40:6 – for (...) { }

attribute	value
$file.name	grid_x.c
$body.num_lines	20
$loop.type	for

Figure 10. The evaluation of pointcut expression results in a table, where each row captures a chain of related join points, whose attributes can be used in actions and conditions.

To access the join points resultant from the evaluation of a pointcut expression, we simply attach the $ operator with the join point identifier. In our example, **$file**, **$function**, **$body** and **$loop** refer to chained join points. We can use these join points and their attributes as part of an action or a condition definition. Join point identifiers enable access to join point attributes, e.g., $file.name refers to the name of the file where a particular $loop join point can be found. In addition, an action can target any element associated with a join point chain, which by default is the last element of the pointcut expression. In the following example, the first action inserts code after the $loop join point, whereas the second inserts code after the $body join point.

```
select function{"main"}.body.loop end
apply
    insert after %{ ... }%;
    $body.insert before %{ ... }%
end
```

Furthermore, multiple pointcut expressions can be joined and associated with a single *apply*, allowing access to multiple branches in the join point hierarchy at once, thus reducing the size of the aspect definition. In particular, LARA supports the (**::**) operator which, when used in an apply definition, joins the results of two pointcut expressions. In particular, the (**::**) operator performs the natural join of two *select* statements. A natural join performs a set of combinations of two join point chains that are equal on their common point element identifiers. Consider the following example where the first pointcut expression (A) refers to all loops in function *f1*, and the second pointcut expression (B) refers to the first statement of the function.

```
A:select function{"f1"}.loop end
B:select function.first end
apply to A::B ... end
```

A	
function	**loop**
f1	a
f1	b

B	
function	**first**
f1	int x;
f2	int q;

A::B		
function	**loop**	**first**
f1	a	int x;
f1	b	int x;

By using the natural join operator (**::**), we are able to apply actions and access attributes in join points that are found in different hierarchical branches (*$loop* and *$first*) of the join point model.

5.3 Examples of Instrumentation

The following two examples illustrate the practical support provided by the LARA aspects.

5.3.1 Timing an application

Developers often need to time their applications, which would require additional code to be inserted in the original application. One can describe this concern using LARA, so that the aspect can be reused in other applications, while reducing code pollution in the original source.

Figure 11(a) illustrates an aspect description which instruments the main function, and adds the necessary code to time the application (see Figure 11(b)). It contains three pointcut expressions. The first, *SF*, selects the first statement of every file. The second, *SM*, selects the first statement of the main function. The third, *SR,* selects the return statements found in the main function. We join all three pointcut expressions to access these join points in one apply section and insert the required code. Note that the pointcut expression identifier *$first* is renamed to *$fstmt* and *$bstmt* to ensure there is no conflict, and that the necessary join combinations are performed.

```
aspectdef timer
  SF: select file.($fstmt=first) end
  SM: select file.
        function{"main"}.($bstmt=first) end
  SR: select file.function{"main"}.return end
  apply to SF::SM::SR
    $fstmt.insert before %{#include <time.h>}%
    $bstmt.insert before %{
            time_t start,end;
            printf("starting timer!\n");
            time (&start);}%;
    $return.insert before %{
        time (&end);
        printf("elapsed time: %.2lfs\n",
        difftime(end,start));}%;
  end
end
```

(a)

```
#include <time.h>
int main() {
  time_t start,end;
  printf("starting timer!\n"); time(&start);
  …
  f();
  …
  time (&end);printf("elapsed time: %.2lfs\n",
  difftime(end,start));
  return 0;
}
```

(b)

Figure 11. Timing applications: (a) aspect that instruments the code to time the application; (b) code of the application after weaving (inserted code is highlighted).

5.3.2 Counting number of loop iterations

Loops can often induce hot-spots in the application, and finding the number of iterations can be useful to determine the applicability of specific transformations. Figure 12(a) illustrates an aspect which instruments every *for* loop using two pointcut expressions. The first pointcut expression selects the loop itself, in which two actions are applied, namely initializing the loop count before the loop and printing the count result after loop execution, respectively. The second selection point selects the first statement of the loop body, where an increment loop count is inserted before that statement. The loop count *uid* attribute returns a unique identifier for every join point, and in this case ensures that every declared variable is unique. An example of the code after weaving is shown in Figure 12(b).

```
aspectdef loop_count
 select loop{type=="for"} end
 apply insert before
        %{int count_[[$loop.uid]] = 0;}%;
      insert after
        %{printf("loop [%s(), %d, %s]:
          %d\n", "[[$function.name]]",
          [[$loop.level]], "[[$loop.loc]]",
          count_[[$loop.uid]]);}%;
 end
 select loop{type=="for"}.body.first end
 apply insert after
        %{count_[[$loop.uid]]++;}%; end
end
```

(a)

```
…{
  int count_01 = 0;
  for (int i = 0; i < N; i++) {
    count_01++;
    int count_02 = 0;
    for (int j = 0; j < M; j++) {
      count_02++;
      ...
    }
    printf("loop [%s(), %d, %s]: %d\n",
           main, 1, "(10,5)", count_02) ;
 }
  printf("loop [%s(), %d, %s]: %d\n",
        main, 0, "(7,2)", count_01) ;
}
```

(b)

Figure 12. Adding loop count for every loop in the application: (a) aspect; (b) woven code.

6. Impact and Experimental Evaluation

In this section we present the experimental results using our compilation and synthesis design flow approach on four real-life applications which have been provided by two industrial partners in the REFLECT project. These applications consist of two audio domain codes, namely an MPEG-2 Audio encoder (MPEG) and a G729 Voice Encoder and two avionics applications, stereo navigation (SN) and 3D Path Planning (3DPP). Both partners have identified a set of concerns and application requirements (e.g., safety in the avionics 3D path planning code) from an industrial perspective which we translated to aspects using the LARA language.

These concerns relate to different stages of the development cycle including: monitoring, fine-tune optimizations, and efficiently mapping the application to the target architecture using schemes such as hardware/software partitioning. As an illustrative example we present in Table 2 a list of concerns for the MPEG Audio encoder application provided by one of the REFLECT industry partners. These concerns span the application development cycle from the analysis phase (with the use of instrumentation to log and monitor specific properties) to the mapping to the target architecture. This, we believe, demonstrates the flexibility and wide scope of our approach.

The monitoring capability enables us to understand runtime behavior, such as (a) the most executed paths in *if-then* constructs, (b) non-variant parameters in specific function calls which enable distinct specializations for the same function, and (c) data ranges and accuracy that can be used to guide word-length optimizations. The use of aspects related to (b) allowed us to identify opportunities for function specializations in all the four applications. For instance, in 3D Path Planning we evaluated three specialized versions of the *griditerate* function, and in the Stereo Navigation we have three possible specializations of a function. These specializations were important to achieve better performance as in most cases, they were related to loop iteration counts, enabling fully loop unrolling and operator strength reduction.

The use of hardware/software partitioning directives allows us to guide the toolchain to map specific functions or code sections to hardware-customized cores. This, in turn, enabled the definition of compilation sequences producing designs that met application requirements and target architecture features.

We next present experimental results using the AOP approach presented in this paper. These experimental results focus on examples of aspects used for application analysis for understanding their dynamic behavior, and on a number of aspects used for performance tuning considering both software and hardware implementations.

Table 2. Examples of aspects applied to the MPEG Audio encoder.

Type of Aspects	Examples
Monitoring (C code insertion)	Monitor range of the variables **z** and **m** in **fsubband** for word-length optimization.
	After word-length optimization, monitors the output variable **s** and variable **y** to validate deviations to their original values.
	Replicate the body of the **fsubband** function with different word-lengths. So, deviation analysis can be done internally.
Specializing	Variables defined as "double" (in functions **add_sub**, **fsubband** and **II_f_f_t**) should be analyzed by the word-length analysis tool to optimize their sizes.
	Convert "double" to "float" data-types in function **II_f_f_t**. A "Code Insertion" aspect is added to monitor the deviations introduced by this transformation.
Mapping and Guiding	Map to hardware functions **add_sub**, **fsubband** and **II_f_f_t**.
	Define specific hardware mapping strategies for **add_sub**, **fsubband** and **II_f_f_t** functions.
	FIFOs, as well as hardware cores for audio I/O, are the hardware blocks necessary in any audio system. These hardware blocks need to interface with the **read_samples** function in the specializing aspect. Aspects can be used to add these hardware blocks without modification of the original code.
	Binding the functions **pow** and **log10** in the **II_f_f_t** and **add_db** to hardware.
	Arrays in functions **fsubband** and **II_f_f_t** mapped to external memories.

6.1 Aspects for Application Code Analysis

Table 3 shows the impact of LARA on the four applications used in our experiments. We consider the following monitoring concerns: measuring branch frequencies, monitoring range (min and max) for variables in the program, monitoring function calls, and timing the application. To assess the impact, we use five metrics: LOC, CDLOC [13], aspectual bloat [14], tangling ratio [14], and the percentage of functions that are affected by the given aspect. We explain each of these metrics next.

The LOC metrics referred in Table 3 are: (a) the number of lines of code in the original application, (b) the number of lines of the woven application, and (c) the number of lines in the LARA descriptions. They provide us with a measure of size for both C and LARA descriptions. For instance, applications SN and MPEG are 9 times larger than 3DPP. On the other hand the timing aspect description is 4 times larger than the call monitoring aspect.

The CDLOC (concern diffusion over LOC) metric indicates the number of switch points where concern-specific code transitions to functional code and vice-versa [13]. We report on the tangling ratio metric, as the ratio between the CDLOC count and the woven code LOC. The tangling ratio gives us a more accurate measure of intermingling. The higher the tangling ratio, the more intermixed is the concern code with the functional code. The lower the tangling ratio, the more localized the concern related code is. The four highlighted aspects (with names ending with 3DPP) in Table 3 target one specific function in the 3DPP application. As these aspects only focus on one code section, their tangling ratio is considerably lower than other wide scope aspects.

Table 3. Aspect metrics for several monitoring concerns: branch frequencies, variable range, function calls, and timing.

Aspect	**App**	**LOC (original C)**	**LOC (woven C)**	**LOC (LARA)**	**CDLOC**	**Tangling Ratio (%)**	**Aspectual Bloat**	**% of affected functions**
FrqBranches3DPP	3DPP	1152	1157	17	10	0.86	0.29	2.17
FrqBranches	3DPP	1152	1187	20	250	21.06	1.75	32.61
FrqBranches	SN	9394	9958	20	1128	11.33	28.20	57.43
FrqBranches	MPEG	8925	9312	20	774	8.31	19.35	56.05
FrqBranches	G729	5835	6094	20	518	8.50	12.95	48.65
RangeExec3DPP	3DPP	1152	1167	25	30	2.57	0.60	4.35
RangeExec	3DPP	1152	1374	23	444	32.31	9.65	39.13
RangeExec	SN	9394	11125	23	3462	31.12	75.26	89.19
RangeExec	MPEG	8925	10747	23	3644	33.91	79.22	84.71
RangeExec	G729	5835	7560	23	3450	45.63	75.00	90.99
CallExec3DPP	3DPP	1152	1159	18	12	1.04	0.39	13.04
CallExec	3DPP	1152	1340	12	188	14.03	15.67	63.04
CallExec	SN	9394	10006	12	612	6.12	51.00	41.22
CallExec	MPEG	8925	10109	12	1184	11.71	98.67	55.41
CallExec	G729	5835	6365	12	530	8.33	44.17	31.53
TimerPC3DPP	3DPP	1152	1185	40	34	2.87	0.83	4.35
TimerPC	3DPP	1152	1533	40	382	24.92	9.53	63.04
TimerPC	SN	9394	10623	40	1230	11.58	30.73	41.22
TimerPC	MPEG	8925	11299	40	2374	21.01	59.35	55.41
TimerPC	G729	5835	6900	40	1066	15.45	26.63	31.53
Average		*5291.6*	*6010*	*24*	*1066*	*15.63*	*31.96*	*45.23*

The aspect bloat measures the efficency of an aspect with respect to the woven code generated. This metric is computed by dividing the number of lines of code that implement a concern by the aspect LOC. If the aspect bloat is less than 1, it means low aspect efficiency as more code was used to write the aspect than the code to implement the concern. In general, a higher aspect bloat means that the aspect has a higher impact factor in the application, and potentially higher reuse.

Table 4 shows the aspect metric results for the following hardware/software partitioning strategy:

```
import gprof_results;

aspectdef PartitionStrategy
  initialize call gprof_results; end
  select function{*} end
  apply map(to: "hardware", id: "virtex5"); end
  condition $function.time > 15 &&
            $function.name != "main" &&
            $function.is_synthesizable end
end
```

which states that all functions that contribute to more than 15% of the total application time and are synthesizable shall be mapped to the FPGA (hardware). The remaining functions are to execute in the PowerPC (software). The toolchain automatically converts to LARA the *gprof* profiling results associating attributes such as timing and number of calls to each function. The Harmonic weaver splits the application into two C sub-applications: one for the PowerPC and another for the FPGA. Each source contains the functions for each partition, in addition to global variables and type definitions that are shared by both partitions, and the code to implement the remote procedure calls. Note that the number of functions mapped to the FPGA, as shown in Table 4, includes not only the functions that run more than 15% of the total application time, but also includes all invoked functions which are part of their implementation.

These results indicate the potential of a LARA aspect to control the entire hardware/software partitioning process without developer intervention. With this approach other partitioning can easily be evaluated without the tedious and error prone manual efforts otherwise required.

Table 4. Aspect metrics extracted when applying LARA aspects for hardware/software partitioning.

App	**LOC** original C	**LOC** woven C	**LOC** LARA	**total # functions**	**# functions mapped to FPGA**	**Aspectual Bloat**	**% of affected functions**
3DPP	1,152	1,246	15	46	7	6.27	15.22
SN	9,394	9,646	15	148	102	16.80	68.92
MPEG	8,925	9,087	15	157	3	10.80	1.91
G729	5,835	5,998	15	111	2	10.87	1.80

6.2 Aspects for Performance Tuning

We now illustrate the use of performance tuning aspects for two specific hot-spot functions, respectively *fsubband* (from MPEG), and *griditerate* (from 3DPP). We report results for a Xilinx Virtex5 FPGA (xc5vfx130t-2ff1738) in terms of the number of FPGA slices and DSP blocks. The software implementations were derived using *ppc-gcc* as backend (w/ option -O3) whereas for the hardware results we used Catapult-C as high-level synthesis tool [15] and the Xilinx ISE 13.1 as back-end tool. For all the results we used fixed-point implementations of the functions and considered the baseline hardware and software implementation as the original untransformed C code.

Figure 13 and Figure 14 illustrate, respectively, the impact of the application of the hardware-oriented aspects on hardware resources[3] and on hardware and software speedups over baseline implementations for different implementations of functions *fsubband* and *griditerate*, using different strategies specified as aspects.

The strategies for *fsubband* included *loopscalar* (B1-B8), *loop-pipelining* (B2-B8), *unroll-and-jam* first nested loops (B3), *fully unroll* first innermost loop (B4), *fully unroll* first innermost loop + *unroll* 16 times of inner second nested loops (B5), *unroll+jam* first nested loops and *unroll* 2 times of inner second nested loops (B6), *unroll+jam* first nested loops and *unroll* 2 times of inner second nested loops (with second index variables) (B7), and fully unrolling of all the 4 loops (B8). The best strategy allows us to achieve a speedup of 10 times over the baseline implementation with only an increase of 9% and 57% of hardware resources (number of slices and DSP blocks, respectively).

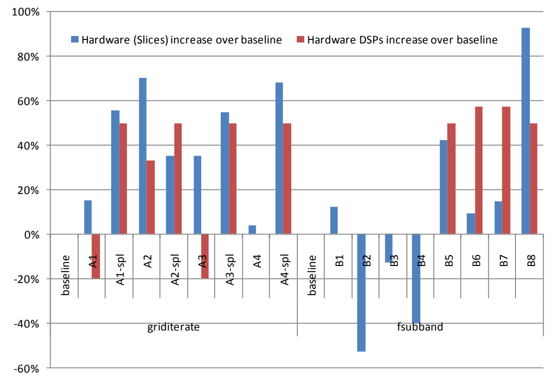


Figure 13. Hardware resources for *griditerate* and *fsubband* considering different strategies over baseline.

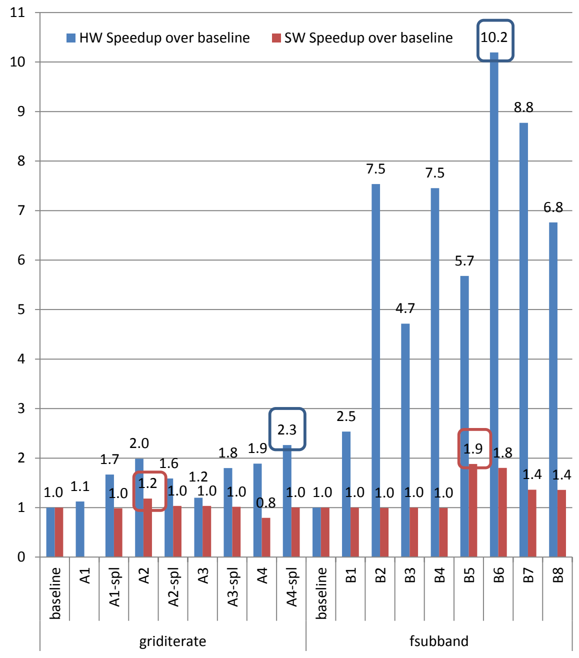


Figure 14. Hardware (HW) and software (SW) speedups over the respective baseline implementations for the *griditerate* and *fsubband* functions considering different strategies.

The strategies for *griditerate* included loop pipelining (A1), specialized versions (A*n*-spl), loop coalescing of the two inner loops (A2), loop unrolling of the innermost loop by two (A3), and data replication in 4 memories (A4). The strategies for *griditerate* result in performance improvements over the reference FPGA hardware implementation of a maximum speedup of 2.26 with an increase of 68% of slices and 50% of DSP blocks.

[3] Xilinx's FPGAs are organized as arrays of programmable slices, interconnect resources, local memories, and DSP components that implement logic functions and local storage structures. For example, on a Virtex-5 FPGA, a 32-bit integer adder requires 5 slices whereas a 32-bit integer multiplier requires 3 DSP blocks. The Virtex-5 FX FPGA also includes two PowerPC processors.

Our approach also allowed us to have two distinct strategies: one used for the best FPGA hardware implementation, and another one used for the best FPGA software implementation (i.e., for the code generated for the GPP). For the software implementation we use one of the PowerPC hardcores of the target FPGA. Figure 14 presents speedups obtained using different strategies. It is clear from the results that the two toolchain flow paths (software and hardware) require different strategies to obtain the best performance. For the *griditerate* function the best strategies are A4-spl and A2 for hardware and software implementation, respectively. For *fsuband*, the best strategies are B6 and B5, for hardware and software implementation, respectively.

6.3 Discussion

Regarding the aspect-related code transformation and analysis metrics, the results reveal high tangling and aspectual bloating ratios of, respectively, 15.6% and 31.9%. These are particularly encouraging as the percentage of affected functions is quite large.

With respect to the software- and hardware-oriented mapping and guiding transformations, the attained performance for the best compilation and synthesis strategy is also very good as the results exhibit very consistent speedups. Developers are able to leverage the compactness of aspect-based mapping strategies and back-end synthesis tools to quickly, and predictably, develop strategies to map applications to hardware/software target architectures.

While we have not yet focused on an evaluation of this aspect-based approach for DSE, these early experimental results do reinforce the claim of its usefulness in the mapping of real applications to complex target architectures.

7. Related Work

We now highlight related work in the areas of Aspect Oriented Programming (AOP) [4][16][17].

AOP has been the subject of intense research over the last decade. AspectJ [18], one of the most widely known AOP languages, is an AOP extension for Java aimed at providing better modularity for Java programs. AspectJ contributes to cleaner and better code by modularizing programs, providing solution for several cross-cutting concerns such as monitoring, logging, debugging, synchronization, and performance. Another example of an AOP language with noticeable success is AspectC++ [19][20], which is an AOP extension to the C++ programming language. Both AspectJ and AspectC++ do not consider join points related to local variables, statements, loops and conditional constructs. For instance, LoopsAJ [21] extends the join point model of AspectJ to allow the direct intervention over loops by adding a new *loop* pointcut (captured with different pointcut expressions), including contextual information used for loop parallelization. @AspectJ [22] is a refinement of AspectJ that allows the specification (using labels) of join points at level of individual statements such as *if* and *while* loops.

Reflecting AOP's growing acceptance, several AOP extensions have been proposed to other languages and domains of applicability. For instance, AspectMatlab [23] provides AOP extensions for MATLAB focusing on array variables and loops, as these are key constructs in scientific applications. AspectMatlab supports the AOP notion of pointcuts (called *patterns*) and advices (called *actions*).

Our AOP-based approach differs from related work in several ways. First, we extend the capabilities associated with types, such as their shapes and precision (as described in our previous work [24][25]). In addition, the join point model allows the specification of pointcuts in virtually all points in a program diminishing the need for labels.

LARA has been inspired by many AOP approaches, such as AspectJ and AspectC++. For instance, during the evolution of LARA we have adopted mechanisms similar to the ones used by AOP approaches based on functional queries [26]. In our approach, pointcut expressions allowed by select sections of the aspect modules are able to define composable select expressions (similar to composable queries) as in [26]. We can associate two or more pointcut expressions to the same advice (apply statement) along with an operator to specify the type of association thus enriching the semantics of the pointcut mechanisms.

We have defined a hardware/software join point model that reflects the need to interface with a potentially wide variety of tools and target embedded computing systems. Lastly, following the notion of patterns and actions in AspectMatlab, our approach also formalizes the concept of strategies as a way to capture and reuse a sequence of program transformations and application mapping choices.

The main drivers of our AOP approach have been the functional requirements elicited by the industrial partners of the REFLECT project [6]. Based on the requirements, LARA includes in the actions associated with join points not only code to be executed (as in AspectJ), but also compiler optimization directives and data and type information about variables. LARA allows powerful dynamic pointcut mechanisms to expose context information about join points.

One of the requirements we faced was the migration of code related to conditional compilation (#ifdef clauses in C) to aspects [27][28]. We have also faced the need of migrating to aspects toolchain directives implemented as C #pragmas. They spread around code artifacts and are commonly used to annotate code with directives for compiler optimizations and code transformations. As their use depends on the target architecture and on the toolchain being used, it is common to have variations for the same application. Aspects mitigate this problem.

In addition, this separation of concerns facilitates the manual exploration of certain compiler properties, as the changes to be evaluated are performed to concentrated code in aspects and not to pragmas spread along the application (as it seems to be a trend [29]). As an example, the Catapult-C version we use [15] allows at least 10 different pragmas. Annotations have severe limitations as they refer to static join points, pollute the code, impose code variations (possibly implemented using conditional compilation mechanisms), and do not allow compiler sequences, while our AOP approach allows semantic and dynamic join points, and join points exposed along compiler sequences.

One of the strengths of our approach is to use AOP to support portability and retargetability. By exposing to aspects concerns such as the ones related to safety and performance requirements, different aspects can lead to the generation of different hardware or hardware/software implementations. This can be conceptually thought as the implementation of portability addressed by Alves *et al.* [30] in the context of software product lines. By exposing to aspects the characteristics of the target architecture, we promote tool-flow adaptability for different architectures. Note that besides code variations we also support AOP-based strategies that allow different implementations by controlling key toolchain stages.

8. Conclusion

This paper presents a novel aspect-oriented programming language named LARA, which provides 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 monitoring, specialization, hardware/software partitioning, and retargeting in the context of multiple programming languages and design flows. The LARA prototype is being evaluated with real-life industrial application C codes. The experimental results provide strong evidence of its usefulness and significance in the mapping of applications to heterogeneous embedded architectures.

We see the flexibility of aspect-oriented approaches, such as the one presented in LARA, as a key programming technology that will enable developers to meet increasingly demanding challenges in developing embedded systems.

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. The authors are grateful to the members of the REFLECT project for their support. Carvalho and Cardoso also acknowledge the support of FCT through grant PTDC/EIA/70271/2006, and Coutinho and Luk acknowledge the support of UK EPSRC.

References

[1] Y. Lam, J. Coutinho, W. Luk, and P. Leong, *Integrated Hardware/Software Codesign for Heterogeneous Computing Systems*, in Proc. of the South. Programmable Logic Conf., 2008, pp. 217–220.

[2] K. Compton, and S. Hauck, *Reconfigurable Computing: a Survey of Systems and Software*, ACM Computing Surveys, 2002, 34(2), pp. 171–210.

[3] T. Todman, et al., *Reconfigurable Computing: Architectures and Design Methods*, IEE Proc. In Computing and Digital Techniques, Vol. 152, No. 2, March 2005, pp. 193-207.

[4] G. Kiczales, *Aspect-Oriented Programming*, in *ACM Computing Surveys (CSUR)*, 1996. 28(4es).

[5] REFLECT, FP7 EU Project: http://www.reflect-project.eu.

[6] J. M. P. Cardoso, *et al., REFLECT: Rendering FPGAs to Multi-Core Embedded Computing*, book chapter in Reconfigurable Computing: From FPGAs to Hardware/Software Codesign, J. M. P. Cardoso and M. Huebner (eds.), Springer, Aug., 2011, pp. 261-289.

[7] J. M. P. Cardoso, *et al.*, *A New Approach to Control and Guide the Mapping of Computations to FPGAs*, in Proc. Int'l Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA'11), July, 2011, CSREA Press, pp. 231-240.

[8] T. Carvalho, *A Meta-Language and Framework for Aspect-Oriented Programming*, Informatics and Computing Eng. MSc Thesis, Univ. of Porto, Faculty of Eng. (FEUP), Porto, Portugal, July 2011.

[9] W. Luk, *et al.*, *A High-Level Compilation Toolchain for Heterogeneous Systems*," in *Proc. IEEE Int'l SOC Conf. (SOCC'09)*, Sept. 2009, pp. 9-18.

[10] ACE CoSy Compiler Development System, http://www.ace.nl/compiler/cosy.html

[11] R. Filman, and D. Friedman, *Aspect-oriented programming is quantification and obliviousness*. In Workshop on Advanced Separation of Concerns at OOPSLA'00, Oct. 2000.

[12] REFLECT Consortium, *LARA Programming Language Specification*, version 1.0 defined as part of deliverable D4.2, Sept. 2011.

[13] E. Figueiredo, et al., *On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework*, in Proc. 12th European Conf. on Software Maintenance and Reengineering, IEEE Computer Society, 2008, pp. 183-192.

[14] C. V. Lopes, *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Nov. 1997.

[15] © Mentor Graphics, Catapult C Synthesis, http://www.mentor.com/esl/catapult

[16] T. Elrad, R. Filman, and A. Bader, *Aspect-Oriented Programming*, in Comm. of the ACM, 44(10), Oct. 2001, pp. 29-32.

[17] G. Kiczales, *et al.*, *Aspect Oriented Programming*, in *Proc. European Conf. on Object-Oriented Programming (ECOOP'97)*, Finland. Springer-Verlag LNCS 1241. June 1997.

[18] J. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. 2003, J. Wiley & Sons, Inc..

[19] D. Lohmann, Olaf Spinczyk. *Aspect-Oriented Programming with C++ and AspectC++*. Tutorial, AOSD'2007, March 13, 2007.

[20] O. Spinczyk, A. Gal, W. Schröder-Preikschat. *AspectC++: An Aspect-Oriented Extension to the C++ Programming Language*. in Proc. 40th Int'l Conf. on Tools Pacific: Objects for internet, mobile and embedded applications, 2002, pp. 53-60.

[21] B. Harbulot, and J. R. Gurd. *A join point for loops in AspectJ*. In Proc. 5th Int'l Conf. on Aspect-Oriented Software Development (AOSD '06). ACM, NY, USA, 2006, pp. 63-74.

[22] M. Poggi. *@AspectJ - An Extension to the AspectJ Join Point Selection Mechanism to Support @Java Annotation Meta-Facility*. Master thesis (in Italian), Università di Genova, Oct. 2009.

[23] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren. *AspectMatlab: An Aspect-Oriented Scientific Programming Language*, in Proc. 9th Int'l Conference on Aspect-Oriented Software Development (AOSD'10). ACM, New York, NY, USA, 2010, pp. 181-192.

[24] J. M. P. Cardoso, J. Fernandes, and M. Monteiro, *Adding Aspect-Oriented Features to MATLAB*, in SPLAT! 2006, Software Engineering Properties of Languages and Aspect Technologies, Workshop affiliated with AOSD 2006, March 2006. Germany.

[25] J. M. P. Cardoso, *et al.*, *A Domain-Specific Aspect Language for Transforming MATLAB Programs*, in Domain-Specific Aspect Language Workshop (DSAL'2010), part of AOSD'10, March 2010.

[26] M. Eichberg, M. Mezini, and K. Ostermann, *Pointcuts as Functional Queries*, in Programming Languages and Systems, W.-N. Chin (Ed.), Springer Berlin/Heidelberg, 2004, pp. 366-381.

[27] V. Alves, *et al.*, *From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration*, In: Aspect-Oriented Product Line Engineering (AOPLE'06), Workshop of the 5th Int'l Conf. on Generative Programming and Component Engineering (GPCE'06), ACM, 2006.

[28] B. Adams, W. Meuter, H. Tromp, and A. Hassan, *Can we refactor conditional compilation into aspects?*, in Proc. 8th ACM Int'l Conf. on Aspect-Oriented Soft. Development, 2009, pp. 243-254.

[29] R. Ferrer, *et al.*, *Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL*, in Proc. LCPC, 2010, pp. 215-229.

[30] V. Alves, *et al.*, *Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming*, in Trans. on Aspect-Oriented Software Development IV, A. Rashid, M. Aksit (Eds.), Springer Berlin / Heidelberg, 2007, pp. 117-142.