A New Approach to Control and Guide the Mapping of Computations to FPGAs

João M. P. Cardoso[3*], Razvan Nane[4], Pedro C. Diniz[2], Zlatko Petrov[1], Kamil Krátký[1], Koen Bertels[4], Michael Hübner[5], Fernando Gonçalves[8], José Gabriel de F. Coutinho[6], George Constantinides[6], Bryan Olivier[7], Wayne Luk[6], Juergen Becker[5], Georgi Kuzmanov[4]

[1]Honeywell International s.r.o., HON, Czech Republic (coordinator)
[2]Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa, INESC-ID, Portugal
[3]Universidade do Porto, Faculdade de Engenharia (FEUP), Portugal
[4]Technische Universiteit Delft, TUD, The Netherlands
[5]Karlsruhe Institute of Technology, KIT, Germany
[6]Imperial College London, Imperial, UK
[7]ACE Associated Compiler Experts b.v., ACE, The Netherlands
[8]Coreworks – Projectos de Circuitos e Sistemas Electrónicos S.A., CW, Portugal

Abstract - *Field-Programmable Gate-Arrays (FPGAs) are becoming increasingly popular as computing platforms for high-performance embedded systems. Their flexibility and customization capabilities allow them to achieve orders of magnitude better performance than conventional embedded computing systems. Programming FPGAs is, however, cumbersome and error-prone and as a result their true potential is often only achieved at unreasonably high design efforts. The REFLECT (Rendering FPGAs to Multi-Core Embedded Computing) project's design flow consists of a novel compilation and synthesis system approach for FPGA-based platforms. Its design flow relies on Aspect-Oriented Specifications to convey critical domain knowledge to optimizers and mapping engines. An aspect-oriented programming language, LARA (LAnguage for Reconfigurable Architectures), allows the exploration of alternative architectures and design patterns enabling the generation of flexible hardware cores that can be incorporated into larger multi-core designs. We are evaluating the effectiveness of the proposed approach for applications from the domain of audio processing and real-time avionics. In this paper we describe the REFLECT approach and present a number of examples and results using REFLECT's compilation and synthesis tools*.

Keywords: FPGAs, Compilers, Aspect-Oriented Specifications, Reconfigurable Computing

* Contact author: João M. P. Cardoso
Universidade do Porto, Faculdade de Engenharia (FEUP)
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
Email: jmpc@acm.org

1 Introduction

Contemporary Field-Programmable Gate-Arrays (FPGAs) are powerful and sophisticated devices able to implement complex high-performance embedded computing systems [1][2]. Customization allows FPGAs to achieve orders of magnitude better performance than conventional processor systems as they can implement directly in hardware specific high-level operations crystallized as custom computing units. As a result, FPGAs are becoming commonplace in embedded systems and even in some cases in high-performance systems.

However, the benefits of FPGA-based systems over traditional systems come at a cost. The large numbers of potential custom functional units, coupled with the many choices of interconnecting these units, make the mapping of computations to these hardware/software architectures a highly non-trivial process. As a result, the mapping of complex applications to these architectures is accomplished by a labor intensive and error-prone manual process. Programmers must assume the role of hardware designers to synthesize or program the various custom hardware units in low level detail, and also to understand how these units interact with the software portions of the application code. Programmers must partition the computation between the code that is executed on traditional processor cores and the code that is to be synthesized in hardware with the consequent partitioning and mapping of data. The complexity of this mapping process is exacerbated by the fact that the custom computing units may internally exhibit different computation models (e.g., data flow, concurrent synchronous processes) and architectural characteristics (e.g., parallelism, customization), or that the various cores might support functional- or data-parallel concurrent execution paradigms.

It is the aim of the REFLECT project [3][4] to develop an approach to help designers achieve efficient FPGA-based heterogeneous multi-core computing systems. Our approach involves combining different areas of research: aspect-oriented specifications, hardware compilation, design patterns and hardware templates. The goal of this project is to develop, implement and evaluate a novel compilation and synthesis approach for FPGA-based platforms. We rely on Aspect-Oriented System Development (AOSD), with foundations on aspect-oriented programming (AOP) [5][6], to convey critical domain knowledge to mapping engines while preserving the advantages of a high-level imperative programming paradigm in early software development as well as programmer and application portability. We leverage aspect-oriented specifications using LARA (LAnguage for Reconfigurable Architectures), a new domain-specific aspect-oriented programming language, to specify complementary information, optimizations, and mapping strategies. The REFLECT design flow has unique characteristics that allow it to both adapt to and meet different non-functional requirements (e.g., safety requirements [7]).

We are evaluating the effectiveness of the proposed approach using real-life applications provided by REFLECT's industrial partners. This evaluation includes the development of two demonstrators: an avionics mission-critical embedded system and an audio encoder. Both these codes raise realistic and demanding challenges that highlight the power and impact of the base techniques and methodologies in the proposed REFLECT approach over traditional design and mapping methodologies.

In this paper we describe the REFLECT design flow [4] and how aspects and strategies are used to map computations to FPGA based systems. In particular, we show experimental results obtained by mapping kernels from two avionics applications, which illustrate strategies suited to meet high-performance requirements.

This paper is organized as follows. Section 2 presents the architecture being currently targeted in REFLECT. Section 3 illustrates the REFLECT design flow and the main tools being developed, used and extended. Sections 4 and 5 describe, respectively, two application case studies and the use of aspects and design patterns in REFLECT. Section 6 presents the results currently achieved when mapping these codes to a REFLECT target architecture. Section 7 presents related work. Finally, Section 8 concludes this paper.

2 REFLECT Target Architecture

Although the REFLECT design flow can target a multitude of reconfigurable architectures, it currently targets an architecture consisting of a general-purpose processor (GPP) connected to Custom Computing Units (CCUs) based on application-specific architectures. Both these components use a shared memory approach possibly connected via data communication channels. The application-specific architectures are implemented with reconfigurable logic (as in reconfigurable fabrics such as FPGAs) and are generated from the C code of the application being compiled.

An example of the target architecture is depicted in Fig. 1 and consists of a GPP, such as a Xilinx MicroBlaze or IBM PowerPC, tightly coupled with a reconfigurable hardware fabric where Custom Computing Units (CCUs) is defined according to application needs. Collectively, the CCUs define a reconfigurable computing system implementing various execution models in space and in time and can consist of specialized hardware templates. The coupling and interface between the processor and the CCUs are inspired on the Molen machine and programming paradigm [8]. We also envision high-end computing systems (akin to HPC systems) that are composed of several of these base reconfigurable systems interconnected using traditional multiprocessor organization arrangements (e.g., bus, hypercube or trees) and logically organized as distributed memory or shared memory heterogeneous multiprocessor systems. From a software-stack perspective, the heterogeneous system is viewed as a co-processor device of a host system. Reconfigurable resources are not exposed to the operating system of the host system. Instead, there is a simple resident "monitor" system responsible for the communication of data and synchronization with the host and/or I/O channels. The development of an operating system is beyond the scope of the REFLECT project.

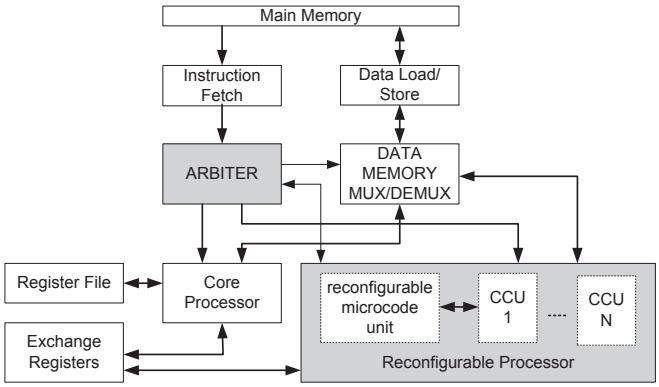

Fig. 1. Block diagram of the target architecture used by REFLECT.

The CCUs and Core processors use a shared memory system and a register file (XREG) to communicate data [8]. For a particular implementation there is a maximum number of CCUs supported by the Molen machine organization and specific FPGA area constraints for CCUs. Dynamic reconfiguration techniques are foreseen for virtualizing the hardware resources. This will allow applications to use during execution more CCUs than the physically available ones.

For prototyping, REFLECT's consortium is using the ML510 Embedded Development Platform which includes a Xilinx Virtex-5 XC5VFX130T FPGA (XC5VFX130T-2FFG1738CES). This FPGA includes two PowerPC 440 processors (PPC440) as hard cores, clocked at the maximum frequency of 400 MHz. The ML510 board consists of several

peripheral interfaces, DRAM memory, and it was selected based on its suitability for prototyping and evaluation of high-performance embedded computing systems.

3 REFLECT's Design Flow

A goal of Aspect-oriented programming (AOP) [5][6] is to improve code modularity by allowing aspects to convey crosscutting concerns. Examples of these include the instrumentation of application code to monitor, debug, and visualize data. When mapping computations to reconfigurable hardware architectures, we are interested in the specification and use of different implementations for the same code. Each of those implementations may take advantage of the specific characteristics of a particular target system, such as memory organization or functional unit capabilities. Aspect modules can thus be used to describe features that a compiler, and other tools in the mapping flow, can use to derive customized solutions, i.e., solutions more suitable to the target architecture and meeting requirements. In this context, we distinguish three main abstractions in the REFLECT compilation/synthesis flow, described in detail in the following sub-sections.

Application Aspects

Application Aspects allow developers to specify application characteristics such as precision representation (e.g., error less than 1E-03), input data rates (e.g., 30 frames per second) or other non-functional requirements such as safety and reliability requirements for the execution of specific code sections/functions. These features act as "requirements" for acceptable design solutions and cannot be easily expressed using common programming languages (such as C). These aspects might be internally decomposed into a number of low-level aspects that guide the REFLECT design flow to generate an implementation which meets the requirements. Some low-level aspects and the ordering of their application can be specified by the user using strategies[1] or can be defined by a Design Space Exploration (DSE) approach. Strategies can thus be seen as rules that force the design flow to apply a specific design pattern.

Design Patterns

Design patterns act as a collection of transformations or "actions" to be used to transform the application code in search for a design implementation with specific features or performance characteristics. For example, an execution time requirement for a specific code section might require the concurrent execution of a specific function. This in turn will require a design pattern or transformation (via the application of strategies) that performs loop unrolling and data partitioning so that data are available to all the concurrently executing units.

[1] The term is used herein in a more generic way than in [9].

Hardware/Software Templates

These templates, which can include a mix of hardware and software implementations, define the "lower" layers of the mapping hierarchy. These templates are characterized in terms of resource usage and number of clock cycles in a specific custom design (e.g., as in FPGAs) as they expose the characteristics of the target resources to the design flow. As an example, the hardware versions of a FIFO or streaming buffer and the software implementation of the same components can be considered hardware/software templates.

Overall, the developer defines, as a first step, the application aspects related to the code at hand, relying on a wealth of existing design patterns and hardware/software templates together with DSE support to find a suitable set of transformations or design patterns that can lead to a specific feasible implementation. The REFLECT compilation flow will benefit from aspects to produce efficient FPGA implementations. This approach is also applicable to other contemporary reconfigurable and non-reconfigurable computing architectures.

In REFLECT we focus on the use of aspects, strategies, and design pattern modules for four types of features:

- SPECIALIZING: Specialization of a design for the particular target 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 the tools in some decisions (e.g., mapping array variables to memories, specifying FIFOs to communicate data between cores, use specific dynamic reconfiguration techniques, use specific fault-tolerance schemes).
- 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.
- RETARGETING: Specification of certain characteristics of the target system in order to make the tools adaptable and aware of those characteristics (i.e., retargetable).

An important component of the aspect-oriented programming model is the notion of a *weaver*. A weaver is a compilation framework component that receives as input the code of the application augmented with the aspect modules, and produces a new version of the code for the application as result of applying the descriptions (rules) in the aspect modules. The aspect modules usually define a *pointcut* and an *advice* [10]. An example of a pointcut and an advice are respectively "find invocations of functions" and "test if array arguments have size greater than 0". In this case, a weaver will insert additional code at each function invocation site to test if array arguments have size greater than zero.

We now describe the overall REFLECT compilation and synthesis design flow. In REFLECT, an input application in C is implemented as a system consisting of one GPP connected

to one or more hardware cores (CCUs), as presented in Fig. 1. Such an application is partitioned for software and hardware components according to developer-provided requirements.

Fig. 2 depicts the main stages of the REFLECT design flow for the generation of hardware and software components. The design flow includes the Harmonic tool [11], which is used as a source-to-source transformation tool. Harmonic is responsible for analyzing and giving hints about code compliance[2], for partitioning the input applications in software/hardware components, for including the communication primitives, and for providing support for code insertion. Aspects related to these transformations are identified by the Aspect Front-End tool and input to Harmonic as Aspect-IR. Harmonic also performs cost estimates for a given platform to assist in the software/hardware partitioning of the input application code. When performing hardware/software partition, the software components are augmented with primitives to communicate data and to synchronize their execution with the hardware components.

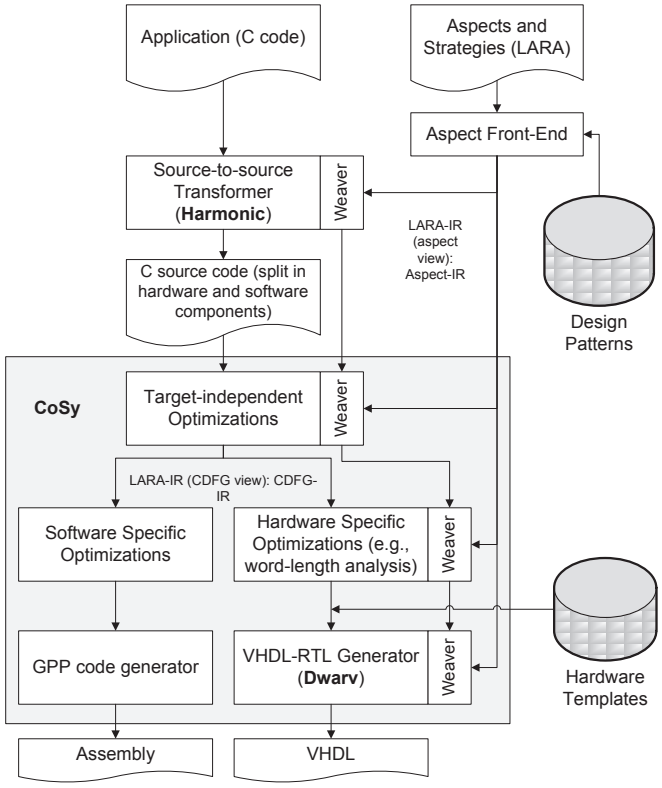

Fig. 2. REFLECT's design flow and its main stages.

The C code output from Harmonic is then input to a CoSy [12] compiler. This CoSy compiler directly invokes the subsequent design flow components, including the weavers to implement some aspects, further target-required optimizations and transformations, and word-length optimizations. Then, the CoSy compiler is responsible for the generation of hardware (hardware components) and RISC code (software component). These components communicate through a common intermediate representation based on a CDFG (Control/Data Flow Graph) represented using CoSy CCMIR (Common CoSy Medium-level Intermediate Representation) and including data-dependences and annotations. This representation is common among the design flow components integrated in CoSy as depicted in Fig. 2.

Strategies, defined as sequences of aspects to be applied, are described in LARA using constructs based on aspect-oriented programming and scripting languages. These strategies enhance DSE via try-and-feedback schemes, implementation of the design patterns and their strategies, and alternative flows for host simulation and target compilation. The LARA-IR carries all information between the components: the transformed and gradually specialized and mapped representation of the application, and all kinds of attributes, not only simple attributes (such as memory spaces of variables) and structured (such as loop-nest information and dependences), but also those that support aspects. At some point in the design flow, the intended partitioning is reflected in the LARA-IR by creating one partition per target architecture and having separate further design flows for each one.

For hardware synthesis, the REFLECT flow uses a tool based on DWARV [13] and integrated in CoSy. As a result, our design flow generates VHDL for the hardware kernels using the same LARA-IR and the same options for arranging the order of transformations as described above. In particular, it applies transformations required to translate a computation from a Von Neumann model of computation to a structural model more suitable for FPGAs. Also, in this phase, the flow implements word-length optimization identified in earlier phases of the mapping. DWARV also implements the weaving phases of the flow what are related to hardware mapping and carries out the DSE for generating high-quality VHDL. To accomplish this, our tool flow is based on a CDFG representation LARA-IR view. This LARA-IR (CDFG view) is then input to DWARV to generate the VHDL descriptions of the hardware modules to be included in the final system as CCUs.

The software components are mapped to a RISC processor core and compiled using CoSy. A further design flow for the software components may include the generation by CoSy of a low-level C representation of the part of the application that should run in the processor, which is fed through a specialized compiler and linker (such as mb-gcc, or ppc-gcc).

Aspect modules, strategies, and design patterns bring to REFLECT's design flow the flexibility and modularity needed to obtain better results and implementations aware of certain concerns. The engines responsible for the application of the concerns described in the aspect modules can take advantage of code refactoring, code transformations (e.g., loop transformations), term-rewriting, etc.

Our approach to maintain aspect modules as primary entities which are not embedded in the application code is

[2] For instance, the VHDL generator used in the back-end of CoSy may not support all C programming constructs.

important to preserve the code's readability and maintenance. This also promotes the reusability of aspect modules and strategies. Multiple aspects and strategies can be applied to the same input application specialized according to the target system organization (e.g., including hardware cores, interface between the GPP and the hardware cores, memories connected to the FPGA, possible precisions). This approach leads to better adaptability of the tools to the specific target and/or non-functional requirements.

In the REFLECT design flow, an aspect includes two main sections: the *select* and the *apply* sections. A *select* section indicates the join points to which the user associates one or more actions specified in the *apply* sections[3]. Our join point model extends traditional join point models of AOP languages such as AspectJ and AspectC++. In our case, join points include system components, code artifacts (loops, functions, variables, assignments, etc.), and code sections (identified by specific pragmas). Each join point artifact has a number of attributes. Those attributes can be used by the actions (specified in the apply sections of the aspect). For instance, a *function* join point includes as attributes, its name, the number of lines, the number of statements, the hardware cost, and the latency. A custom computing unit (CCU) join point may have as attributes the clock frequency, the maximum hardware area, etc. Actions can depend on the attributes for a particular join point, and they can define values for those attributes. Most attributes are defined by the stages of the REFLECT design flow.

Fig. 3 depicts an aspect that can be used to map a function with name "fir" to hardware (in our example to a CCU [8]). This aspect invokes an aspect named "strategy1" which includes optimization rules, user's 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.

```
aspectdef maxmizePerformance
  sel1 select: *.function{name="fir"} // specification of pointcut
  apply to Sel1: map to hardware      // map action
  apply to Sel1: call strategy1       // call action
  constraints:                        // constraints
    define function{"fir"}.arg{output}.noise_power <= 1e-3;
    define function{"fir"}.arg{input}.range = -40..120;
  end
end
```

Fig. 3. Example of an aspect specifying non-functional requirements.

Each type of action is associated to a specific stage in the REFLECT design flow. For instance, the *optimize* action is associated to the CoSy compiler instance and includes compiler optimizations such as loop unrolling, scalar replacement, loop fusion, loop fission, code hoisting, word length analysis, and data-type conversion.

Aspect actions (apply section) can be of different types. Table 1 presents the current type of actions being considered. These actions include mapping and optimization directives as well as directives to specify the insertion of code in specific join points (used for monitoring and instrumenting) to define properties and to instruct tools to report information (e.g., values of attributes).

Table 1. Current keywords associated to actions.

Action (keyword)	Description
insert	insertion of code
report	instructs the tools in the REFLECT design flow to report information
optimize	instructs the tools for specific optimizations, including code transformations
map	Instructs the tools to map computations and data structures to specific hardware components
define	defines properties that can be used by the tools
call	invoke other aspects

4 Case Studies

We now describe opportunities for the application of various Aspects described above to the hot-spots of two applications from the avionics domain: 3D Path Planning and Stereo Navigation. In this section we briefly describe their computations, and the following section presents experimental results of the application of a set of high-level code transformations guided by the use of Aspects.

4.1 3D Path Planning

The 3D Path Planning core computation defines a 3D path $r(t)$ between the current vehicle position and required goal position, using Laplace's equation (see, e.g., [15][16]). It solves Laplace's equation in the interior of a 3D region, guaranteeing no local minima in the interior of the domain, leaving a global minimum of $v(r) = -1$ for r on the goal region, and global maxima of $v(r) = 0$ for r on any boundaries or obstacle. A path from any initial point $r(0)$, to the goal, is constructed by following the negative gradient of the potential, v.

Fig. 4 illustrates the computational contribution of the main 3D Path Planning functions to the global execution time on a PPC440 core (at 400 MHz) embedded in a Xilinx Virtex5 FPGA. The iteration steps represent over 90% of the global execution and are performed by the *gridIterate* function.

A possible code implementation for the *gridIterate* function is depicted in Fig. 5 where, for simplicity, details such as global variables definition, initialization and functions, are omitted. This function uses a 3D matrix representing an obstacle map (array *obstacle*) and outputs a 3D matrix representing the potential matrix (array *pot*).

[3] The select and apply are conceptually equivalent to the *pointcut* (set of join points) and *advice* in AspectJ [10] and have been previously used in the context of an aspect language for MATLAB [14].

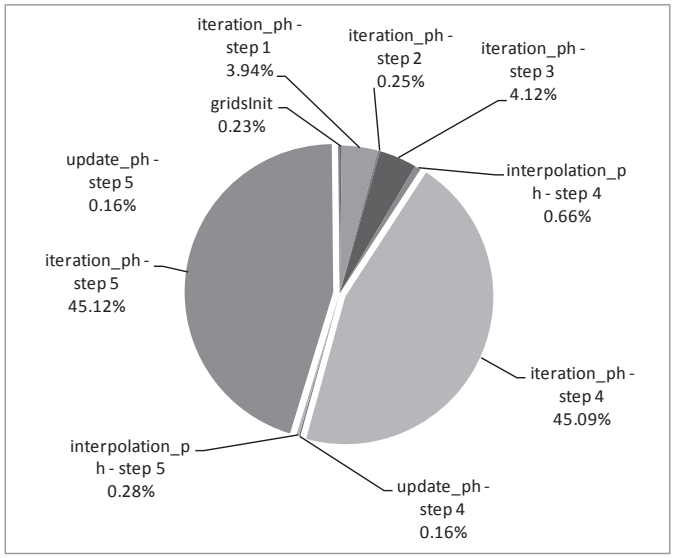

Fig. 4. Contribution of the main 3D Path Planning functions to the global execution time (obtained by using hardware timers).

```
#define ITER_STEPS_NUM ...
void gridIterate(int* obstacles, float* pot)  { …
 for (it = 0; it < ITER_STEPS_NUM; it++) {
  for (i = 1; i < (X_DIM - 1); i++) {
   for (j = 1; j < (Y_DIM - 1); j++) {
    for (k = 1; k < (Z_DIM - 1); k++) {
     val = obstacles[i][j][k];

     if (val == 1) pot[i][j][k] = POTENTIAL_ZERO;
     else if (val == -1) pot[i][j][k] = POTENTIAL_ONE;
     else {
       acc = (accType)pot[i-1][j][k]+
             (accType)pot[i+1][j][k] +
             (accType)pot[i][j-1][k]+
             (accType)pot[i][j+1][k] +
             (accType)pot[i][j][k-1]+
             (accType)pot[i][j][k+1];
       pot[i][j][k] = FIX_CORRECT(acc * SCALE);
     }}}}}}
```

Fig. 5. Function *gridIterate* C code from the 3D Path Planning application.

4.2 Stereo Navigation

The Stereo Navigation (*StereoNav*) application is intended for airplane localization when the GNSS (Global Navigation Satellite System) used in airplanes is temporarily unavailable. The idea of the application is that from two independent images derived from cameras, looking in approximately the same direction, features can be extracted (dominant entities in the image are invariant to rotation and translation). Using two cameras taking simultaneous images allows for localization of the features in 3D-space. The main components of the algorithm include: Debayering (optional), Rectification, Feature extraction, Feature matching, 3D reprojection, and Robust pose estimation and refinement.

Fig. 6 illustrates the contribution of the main StereoNav functions to the global execution time when executing the application in the PPC440 core (at 400 MHz) in the Xilinx Virtex5 FPGA. We used hardware timers to measure the execution time of each function. The core computation of the *StereoNav* application is presented in function *harrisTile_model_step* (identified in Fig. 6 as "do_tile") and consists of a sequence of 8 convolutions using two kinds of *conv* function (*ConvVBConst* and *ConvVBRepl*). A section of the C code of the *ConvVBConst* function is depicted in Fig. 7.

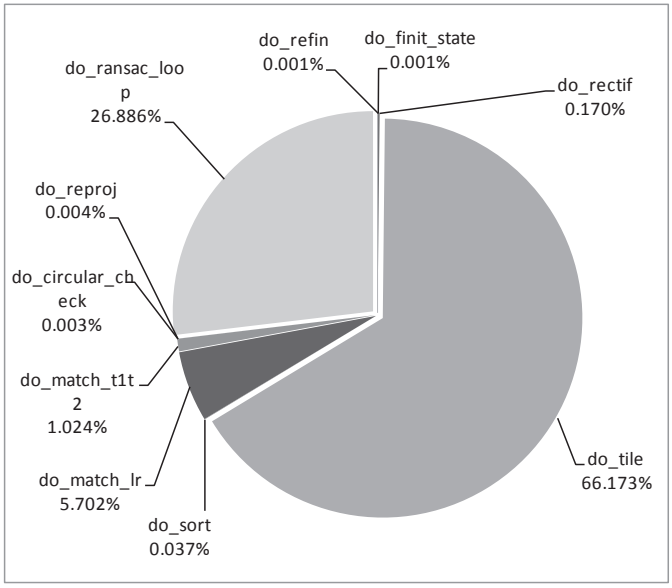

Fig. 6. Contribution of the main *StereoNav* functions to the global execution time (obtained by using hardware timers).

```
void ConvVBConst (..) { ...
for (IDXB_1U=SSTART_1U; IDXB_1U<=SEND_1U; IDXB_1U++){
  …
  for (IDXB_0U=SSTART_0U;IDXB_0U<=SEND_0U;IDXB_0U++) {
    … acc = 0.0F;  …
    for (HIDXA_1U=0;HIDXA_1U<=HEND_0_1U;HIDXA_1U++) {
      for (HIDXA_0U=0;HIDXA_0U<=HEND_0_0U;HIDXA_0U++) {
        acc = u[IDXALIN_0U] * h[buf1Idx] + acc;  …
      } IDXALIN_0U = (uDims[0U] - hDims[0U]) + IDXALIN_0U;
    } y[IDXBLIN_0U] = acc;
  }
 }
... // second part of the convolution
  //(with 5 FORs nested and a function call in the innermost loop
}
```

Fig. 7. Part of the C code of the *ConvVBConst* function.

5 REFLECT Approach

This section illustrates the use of aspects, design patterns and hardware/software templates for the two case studies described in the previous section.

5.1 3D Path Planning: gridIterate

For the *gridIterate* function we consider the optimizations and strategies presented in Table 2 and Table 3, respectively. For this case study we focus on data conversions from floating-point to fixed-point, partial loop unrolling of the innermost loop, and multi-dimensional arrays transformed to uni-dimensional arrays. As the data elements defining *obstacles* have values in the set {-1, 0, 1} and that the *pot* data represent real values in the range [0, 1], scaling analysis can result in an optimized fixed-point representation.

Table 2. Optimizations considered for *gridIterate*

Transf.	Description
T 1.1	Float to fixed-point representation
T 1.2	Unroll innermost loop by 2
T 1.3	Shift by powers of two promoted to wires
T 1.4	Linearization of multi-dimensional arrays.
T 1.5	Array indexing transformed as wire concatenation and wiring component
T 1.6	Code motion (loads moved from if-else conditions)

Table 3. Strategies considered for *gridIterate*

Strategy Name	Transformations					
	T1.1	T1.2	T1.3	T1.4	T1.5	T1.6
gridIt-baseline			✓	✓		
gridIt-fixed1	✓		✓	✓		
gridit-fp1		✓	✓	✓	✓	✓
gridit-fixed2	✓	✓	✓	✓	✓	✓

To convert from multi-dimensional to uni-dimensional arrays, an index such as [i][j][k] is translated to (i*Y_DIM + j)*Z_DIM + k allowing the subsequent application of operator strength reduction on the calculations for the indexing of the array variables as well as concatenation of addressing bits when the various array dimensions are aligned at specific power-of-two address boundaries.

A transformation to the *gridIterate* function considers multi- to uni-dimension transformation, code motion, and the use of a macro that can be implemented as a concatenation of wires to calculate the index of the arrays *pot* and *obstacles*. The code motion is applied based on the following explanation. In order to decrease the number of references to the *pot* array variable, the writes to *pot[i][j][k]* existent in all branches of the *if-else* construct in the code can be moved to after the *if*. This transformation also allows the parallel loads of *obstacles* and *pot* data when the two arrays are bound to different memories or to a multi-port memory. The code motion of the accesses to *pot* allows earlier scheduling of *pot* data loads. If the innermost loop is unrolled twice, we increase the impact of pipelining memory accesses, and we reuse a load to *pot* thus reducing the number of loads per two k-loop iterations.

The mapping of functions to hardware can be guided by the user through aspects. Fig. 8 illustrates a generic aspect to map a given function to a CCU identified by an input id. For instance, by associating a specific instance of this aspect as

```
map2hardware("gridIterate", 1)
```

the *gridIterate* function will be mapped to a CCU of the target architecture identified by "1". Further, the user may use conditions to make an action dependent on the value of certain attributes. For instance, the use of

```
condition: $function.no_lines < 500
```

in the aspect in Fig. 8 instructs the weaving process to map a function to hardware only if the function is less than 500 lines of code long (attributes as hardware cost can also be used).

```
aspectdef map2hardware(string $name, int $id=1)
  select A: function{name=$name}
  apply to A: map to hardware(ccu.id=$id)
end
```

Fig. 8. An aspect with an action to map a function to hardware.

5.2 Stereo Navigation: Convolutions

For the *convolution* functions we consider the optimizations and the strategies presented in Table 4 and Table 5, respectively. The convolution functions *ConvVBConst* and *ConvVBRepl* include invocations to the functions *PadBConst* and *PadBRepl*, respectively. For this second case study we consider scalar replacement, function inlining, and the specialization of the convolution functions according to the calls. This specialization is mainly dedicated to the elimination of loop headers for loops with only one iteration, as well as to the unrolling of innermost loops when their number of iterations is less than or equal to three.

Table 4. Optimizations considered for the convolution functions.

Transf.	Description
T2.1	Scalar replacement
T2.2	Function inlining
T2.3	Specialization of each call to conv
T2.4	Loop header elimination
T2.5	Loop unrolling of innermost loops with number of iterations <= 3

Table 5. Strategies considered for the convolution functions.

Function	Strategy	Transformation				
		T2.1	T2.2	T2.3	T2.4	T2.5
ConvVBConst	stg01	✓	✓			
	stg02	✓	✓	✓		
	stg03	✓	✓	✓		✓
ConvVBRepl	stg04	✓	✓			
	stg05	✓	✓	✓		
	stg06	✓	✓	✓	✓	
	stg07	✓	✓	✓	✓	✓

Fig. 9 illustrates the LARA specification of a strategy that considers function inlining, loop unrolling, and function specialization. The use of *section* (e.g., *section{"l1"}*) in the select sections of the aspects refers to specific code sections identified by pragmas included by the user in the code as *#pragma joinpoint section="l1"*. Note, however, that this is indicative and the final syntax and constructs of LARA may be slightly different.

```
import inline1;
import unroll1;

aspectdef Const_Config1
  select A: function{"harrisTile_model_step"}.section{"l1"}.
            call{"ConvVBConst"}.body;
  apply to A:
    define{$ sEnd[1U]=94, $ sEnd[0U] =94, $hEnd_0[1U]=3;
            $hEnd_0[0U] =3,  $numBSec =8, $ sEnd_0[1U] =94,
            $ sEnd_0[0U]=94, $hEnd_1[1U] =3,$ hEnd_1[0U] =3}
   optimize specialize();
  end
end
aspectdef Repl_Config1
   select A: function{"harrisTile_model_step"}.section{"l2"}.
            call{"ConvVBConst"}.body;
   apply to A:
      define{...}
      optimize specialize();
   end
end
aspectdef Repl_Config2
   select A: function{"harrisTile_model_step"}.section{"l3"}.
             call{"ConvVBConst"}.body;
   apply to A:
       define{...}
      optimize specialize();
   end
end

call unroll1("ConvVBConst");
call unroll1("ConvVBRepl");

call inline1("PadBConst");
call inline1("PadBRepl");

// two imported aspects:
aspectdef inline1(String $name) // inline functions identified bt
$name
  select: function{$name};
  apply: optimize inline();
end

aspectdef unroll1(String $name) // unroll loops if the number of
iterations is <=3
   select A: function{ $name}.loop{*};
   apply to A,B:  optimize loop_unrolling($loop);
   condition: $loop.no_iterations <= 3
end
```

Fig. 9. Examples of aspects and possible strategy for the *harrisTile_model_step* function.

6 Experimental Results

We apply the strategies outlined in Section V to the functions described in Section IV. As our design flow is not yet fully automated, the results presented here correspond to the manual application of the described aspects and strategies. We consider software versions of the functions and compare the results of running them on the PPC440 at 400 MHz against hardware versions obtained by the DWARV compilation and synthesis flow. Unless otherwise stated, the software versions are generated with the gcc compiler using the -O3 compilation option. The hardware versions are clocked at 200 MHz.

3D Path Planning: gridIterate

The use of floating-point data types for the *gridIterate* (*gridIt-baseline* and *grid-fp1*), single precision in this case, favors the use of dedicated hardware implementations. With respect to floating-point solutions, the hardware implementations achieve speedups of 2.15× and 2.83× over the software related versions for *gridIt-baseline* and *gridIt-fp1*, respectively. In the case of the fixed-point solutions (*gridIt-fixed1* and *grid-fixed2*), the hardware implementations achieve speedups of 1.05× and 5.56× over the software solutions.

Considering the FPGA resources used for different hardware implementations of the same function (*gridIterate*) the strategies used for *gridIt-fixed1* and *gridIt-fixed2* imply more hardware resources due to the presence of a 64×64 bit multiplication in the fixed-point multiplication vs. the presence of a 23×23 bit multiplication for the single precision floating-point version (*gridIt-baseline* and *gridIt-fp1*). This is reflected in the use of 2.2× the number of DSP48 and 1.23× the number of slices. The last two strategies (correspondent to *gridIt-fp1* and *gridIt-fixed2*) achieve implementations with more slices than the one using the strategy considered by *gridIt-baseline* and *gridIt-fixed1*. This is due to the fact that *gridIt-fp1* and *gridIt-fixed2* consider loop unrolling of the innermost loops by a factor of 2.

Stereo Navigation: Convolutions

For the function *ConvVBRepl* of the Stereo Navigation application, the use of strategy stg07 allows a speedup of 1.30× by the FPGA design over the software version with the same strategy. For *ConvVBConst* the FPGA design achieves speedups of 2.31× and 2.54× over the best non-specialized software implementation considered (PPC –O3) and non-specialized FPGA implementation, respectively.

The use of strategies stg05 and stg06 in the *ConvVBRepl* functions leads to a decrease in slices of 32.39% and 44.33%, respectively. For *ConvVBConst* the number of slices decreases by 7.62% when using stg03 vs. stg01 for similar clock frequencies.

7 Related Work

Compiling high-level programming languages to FPGAs is a topic that has been extensively addressed by academia and industry (see, e.g., [17] for a survey of representative approaches). However, it is understood that, due to the large gap between software and hardware, compilers for FPGAs still have a long way to go before being able to generate efficient customized architectures for complex applications. In addition, the hardware to be generated depends on non-functional requirements, which are not embedded in the code of the application and result in extensive work by the designer to explore options and to modify the code of the application.

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

Recent efforts to map computations to FPGA-based systems include the hArtes tool chain [17]. hArtes also includes as a source-to-source transformation stage the Harmonic [11] tool, and as a hardware compiler a previous version of DWARV [13]. However, the hArtes approach supports neither an aspect-oriented approach nor strategies and design patterns.

8 Conclusions

This paper presented part of the REFLECT project's approach to a design flow targeting FPGA systems. At the core of our approach is a new programming language, named LARA, allowing users to specify aspects and strategies (reflecting design patterns) that guide the design flow to meet desired non-functional requirements.

Specifically, in this paper we focused on the description of aspects and strategies to two critical functions from two avionics applications: Stereo Navigation, and 3D Path Planning. We presented experimental results of the application of selected aspects and the corresponding strategies. The results highlight the modularity and reusability of aspects and design patterns in the proposed approach, thus providing early evidence that this approach can lead to a substantial cost decrease of code maintenance while promoting design space traceability.

9 Acknowledgment

This work is 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 authors and do not necessarily reflect the views of the European Community. The authors are grateful to all team members of the REFLECT project for their help and support.

10 References

[1] S. Hauck, and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation (Systems on Silicon)*, Morgan Kaufmann, November 16, 2007.

[2] M. Gokhale, and P. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 1st Edition, Dec., 2005.

[3] REFLECT website: http://www.reflect-project.eu.

[4] 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*, Springer (to appear).

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect Oriented Programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, Finland. Springer-Verlag LNCS, vol. 1241, June 1997.

[6] G. Kiczales, "Aspect-Oriented Programming," in *ACM Computing Surveys (CSUR)*, special issue: position statements on strategic directions in computing research, 1996. 28(4es).

[7] Z. Petrov, K. Krátký, J. M. P. Cardoso, and P. C. Diniz, "Programming Safety Requirements in the REFLECT Design Flow," in *IEEE 9th Int'l Conference on Industrial Informatics (INDIN'2011)*, Caparica, Lisbon, Portugal, July 26-29, 2011 (to appear).

[8] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The Molen Polymorphic Processor," in *IEEE Transactions on Computers*, Nov. 2004, Vol. 53, Issue 11, pp. 1363-1375.

[9] R. Lämmel, E. Visser, and J. Visser, "Strategic programming meets adaptive programming," In *Proc. of the 2nd Int'l Conference on Aspect-Oriented Software Development (AOSD '03)*, Boston, Mass., March 17-21, 2003. ACM, New York, NY, USA, pp. 168-177.

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

[11] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A High-Level Compilation Toolchain for Heterogeneous Systems," in *Proc. IEEE International SOC Conference (SOCC'09)*, Sept. 2009, pp. 9-18.

[12] ACE CoSy compiler development system, http://www.ace.nl/compiler/cosy.html

[13] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis, "DWARV: DelftWorkbench Automated Reconfigurable VHDL Generator," in *Proc. of the 17th Int'l Conference on Field Programmable Logic and Applications (FPL'07)*, Aug. 2007, pp. 697–701.

[14] J. M. P. Cardoso, P. Diniz, M. Monteiro, J. Fernandes, and J. Saraiva, "A Domain-Specific Aspect Language for Transforming MATLAB Programs," in *Domain-Specific Aspect Language Workshop (DSAL'2010)*, part of the 9th Int'l Conference on Aspect-Oriented Software Development (AOSD'2010), March 15-19, 2010.

[15] C. I. Connolly, J. B. Burns, R. Weiss, "Path planning using Laplace's equation," in *Proc of IEEE Int'l Conference on Robotics and Automation*, Cincinnati, OH, USA, May 1990, vol. 3, pp. 2102-2106.

[16] K. P. Valavinis, T. Herbert, R. Kollura, and N. Tsourveloudis, "Mobile Robot Navigation in 2-D Dynamic Environments Using an Electrostatic Potential Field," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, issue 2, March. 2000, pp. 187-196.

[17] J. M. P. Cardoso, P. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A Survey," in *ACM Computing Surveys (CSUR)*, Vol. 42, Issue 4, Article 13 (June 2010), 65 pages.

[18] K. Bertels, V. Sima, Y. Yankova, G. Kuzmanov, W. Luk, J. Coutinho, F. Ferrandi, C. Pilato, M. Lattuada, D. Sciuto, and A. Michelotti, "HArtes: Hardware-Software Codesign for Heterogeneous Multicore Platforms," in *IEEE Micro*, 30(5): 2010, pp. 88-97.