2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

Specifying Compiler Strategies for FPGA-based Systems

João M. P. Cardoso[1], João Teixeira[1], José C. Alves[2], Ricardo Nobre[1,3], Pedro C. Diniz[3], José G. F. Coutinho[4], and Wayne Luk[4]

[1] Universidade do Porto
Faculdade de Engenharia
Dep. de Engenharia Informática
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jmpc@acm.org

[2] INESC-TEC, Univ. do Porto
Faculdade de Engenharia, DEEC
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
ee06128@fe.up.pt
jca@fe.up.pt

[3] INESC-ID,
Rua Alves Redol 9
1000-029 Lisboa, Portugal
ricardo.nobre@fe.up.pt
pedro@esda.inesc-id.pt

[4] Department of Computing
Imperial College London
180 Queen's Gate
London SW7 2BZ, UK
gabriel.figueiredo@imperial.ac.uk
w.luk@imperial.ac.uk

Abstract—**The development of applications for high-performance Field Programmable Gate Array (FPGA) based embedded systems is a long and error-prone process. Typically, developers need to be deeply involved in all the stages of the translation and optimization of an application described in a high-level programming language to a lower-level design description to ensure the solution meets the required functionality and performance. This paper describes the use of a novel aspect-oriented hardware/software design approach for FPGA-based embedded platforms. The design-flow uses LARA, a domain-specific aspect-oriented programming language designed to capture high-level specifications of compilation and mapping strategies, including sequences of data/computation transformations and optimizations. With LARA, developers are able to guide a design-flow to partition and map an application between hardware and software components. We illustrate the use of LARA on two complex real-life applications using high-level compilation and synthesis strategies for achieving complete hardware/software implementations with speedups of 2.5× and 6.8× over software-only implementations. By allowing developers to maintain a single application source code, this approach promotes developer productivity as well as code and performance portability.**

Keywords-aspect-oriented programming; compilation; strategies; hardware synthesis; FPGA; hardware acceleration

I. INTRODUCTION

The mapping of applications expressed in high-level imperative programming languages to FPGA-based heterogeneous reconfigurable computing architectures is a highly complex and error-prone process. This difficulty is further exacerbated by the variety of tools and computing paradigms required to target these architectures as well as to satisfy non-functional requirements (NFRs). For instance, a design solution that is suited for inputs with specific data rates may simply not be feasible for other input settings. NFRs, such as the ones related to reliability and safety, performance, and energy consumption, are commonly out of the scope in existing tools, or cannot be expressed using current high-level programming languages.

These factors lead to very long and error prone design methods, in practice forcing developers to settle for sub-optimal design solutions, given the sheer size of the corresponding design-space-exploration (DSE). Even when specific *pragma*-based interfaces exist, the developer is confronted with the unpleasant prospect of having to annotate the source code. As the system evolves, these annotations and code specializations invariably lead to code obfuscation and to code more difficult to maintain. As computing systems in general, and reconfigurable architectures in particular, continue to increase not only in size but also in heterogeneity, so will the effective mapping of computations to these new architectures become prohibitively complex and brittle. In particular, the amount of effort required to achieve optimized implementations and to maintain these codes will be exceedingly high.

In this paper we describe a novel approach for developing FPGA-based designs, where the code capturing the main functionality of the application (in C) is decoupled from NFRs. Strategies to meet NFRs are specified in LARA [1], a domain-specific aspect-oriented programming (AOP) [2] language. By decoupling the application code from non-functional specifications, we preserve a clean source description of the computations which can be maintained separately while taking advantage of a wealth of program and mapping transformations. LARA provides the means of codifying strategies that capture, control and unify a wide range of schemes, including: (a) hardware-software partitioning, (b) instrumentation and monitoring, (c) code specialization, and (d) code optimizations. In addition, LARA provides a unified mechanism to convey to the underlying tools (using a common tool-independent interface) system attributes and NFRs that would otherwise be inaccessible by the design-flow. Our work is further facilitated by three distinct LARA features: (1) LARA descriptions can be parameterized to capture design-patterns and templates and used in the context of DSE, (2) code elements can be annotated in LARA with arbitrary information from third-party tools, such as profilers, that drive DSE, and (3) LARA can control the design-flow and can be extended to control other parts of the system at runtime.

In summary, our contributions include:

1. a design-flow based on LARA to map applications into heterogeneous multi-core FPGA platforms.
2. a set of compiler strategies defined as sequences of program transformations and data mapping directives written in LARA that can be reused to generate efficient designs for reconfigurable systems.
3. an evaluation of our approach with complete implementations of two real-life applications.

While in this work we use Catapult-C [3] to synthesize hardware components, our LARA-based design-flow is completely agnostic to the back-end and front-end tools used. To target other C-to-HDL tools, we would simply need to revise the weavers to set the appropriate interfaces, such as a specific set of C *pragmas* or scripts.

Overall, the results and experiences presented in this paper lead us to believe that an AOP [2] approach such as LARA offers a valuable mechanism to promote both performance and code portability, while enhancing design reuse in the face of current and future dynamic architecture landscapes.

The remainder of this paper is organized as follows. Section II describes the design-flow, LARA, and the target architecture used. Section III describes the case studies used for evaluation of our approach. Section IV presents examples of compiler strategies and shows how they are specified in LARA. Section V presents the results achieved and Section VI focuses on the most relevant related work. Finally, Section VII concludes the paper.

II. DESIGN-FLOW AND TARGET ARCHITECTURE

We now describe the overall design-flow, highlighting its main components and explaining how LARA exploits the design-flow to derive embedded system solutions.

A. *Design-Flow: From Requirements to Aspects*

Figure 1 shows a LARA-based design-flow. It requires two types of descriptions: (1) C sources for the main application functionalities, and (2) LARA description files which capture NFRs in the form of aspects and strategies. Aspects allow developers to convey specific application attributes such as data-precision representations, input data rates, or even reliability requirements, in a way that is completely decoupled from the application source. Developers can also control specific transformations in combination with attributes of selected program constructs to define (in a parametric fashion) powerful compilation and mapping strategies. Based on the feedback from compilation and synthesis processes, developers (or a DSE module) can adjust strategies by changing the values of specific attributes or the sequence of transformations to be applied, to effectively navigate the design space in search of solutions that directly or indirectly satisfy the desired application requirements.

The current design-flow implementation is based on a static compilation flow and is structured as follows:

1. First, LARA source files are compiled to a single Aspect-IR file using the LARA front-end tool. Aspect-IR is an intermediate representation of LARA in XML. It is more verbose than LARA, but it is structured in a way to facilitate the parsing and the weaving of aspects and strategies described originally in LARA.
2. Next, all the weavers in the design-flow are executed in sequence. Each weaver is capable of performing a specific set of actions and receives as inputs: the application (in textual or intermediate format) and the Aspect-IR. The output of each weaver, which is the input for the next weaver in the sequence, is the woven application and a revised Aspect-IR, e.g., without aspect elements no longer applicable or valid.
3. The final stage involves compiling and linking object codes and bitstreams into a single binary.

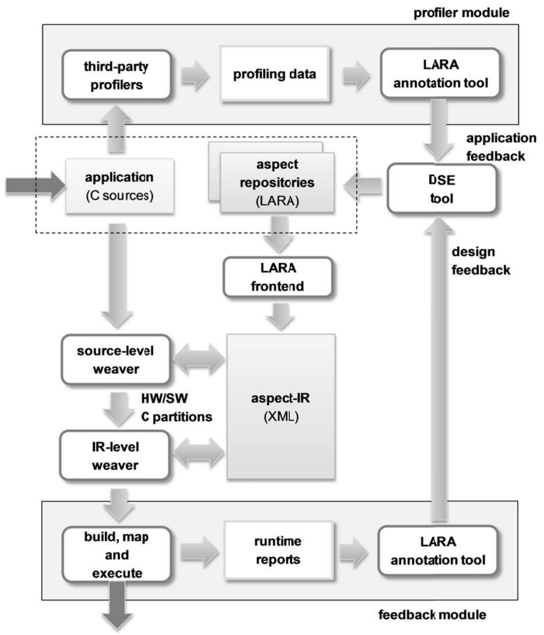


Figure 1. LARA based design-flow.

Our current design-flow includes weavers that process the application at different stages of the compilation chain. Figure 1 presents two weaver components:

Source-level weaver: this weaver performs actions at C source-level. It receives C as input, generates C as output, and it is responsible for two actions: source-level instrumentation and hardware/software partitioning [4]. Aspects processed at this stage can exploit the fact that there is no code lowering, and transformations preserve as much as possible the original structure. When hardware/software partitioning is sought, the weaver generates C partitions that can be compiled separately for each component. The source-level weaver also inserts communication primitives that realize remote procedure calls between software and hardware components. Strategies at this level map code functions to FPGAs using cost estimation models based on code features and on profiling information.

IR-level weaver: this weaver operates on the intermediate representation of the application and provides more opportunities for aggressive optimizations. LARA aspects control optimizations, such as loop transformations. In particular, target-independent optimizations are applied first, followed by software or hardware specific optimizations according to the target for each C partition (generated by the source-level weaver).

Our design-flow also allows the definition and execution of DSE strategies by supporting an interface where the results of external tools can be captured to drive these strategies, and therefore lead to more efficient designs. In particular, we developed tools and interfaces providing:

Profiling information: this information is collected using a third-party profiler tool, such as *gprof*. The profiling results are captured in LARA using a LARA annotation tool, and then imported by other LARA sources, thus influencing the outcome of a particular strategy.

Feedback information: this information can be collected from design-flow stages, such as place-and-route reports from FPGA vendor tools, and can be subsequently used to drive multiple iterations of the toolchain as part of a DSE process. In particular, a simple tool is used to convert reports generated by backend tools (e.g., Xilinx XST) into LARA, to be then imported and used by existing strategies.

As referred before, in this paper we use Catapult-C [3] as the design-flow high-level synthesis tool. The control of Catapult-C is performed by a weaving phase that automatically translates LARA actions for hardware synthesis into specific *pragmas* added to the C code input to Catapult-C.

We now highlight some important features of LARA.

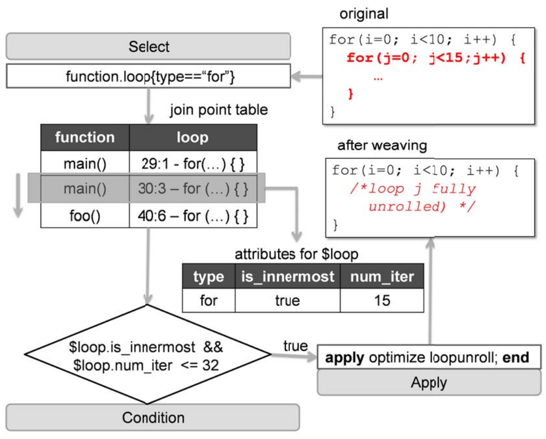
B. *LARA Language*

LARA is an AOP language [1] allowing the specification of NFRs, such as ones related to performance and fault-tolerance, and optimization strategies, in a way decoupled from the original application source code. To promote this separation of specifications (concerns), we rely on weavers that automatically generate an augmented application based on the original code and LARA sources.

LARA descriptions are structured as aspect definitions. Each aspect implements a cross-cutting concern (or part of a concern), and consists of three main sections, namely: *select*, *apply* and *condition*. The *select* section allows developers to define pointcut expressions that capture join points, e.g., parts of the program (function definitions, statements) which one wishes to act upon. The *apply* section defines actions to be applied to each join point resultant from a select query. The woven application is therefore the result of a set of actions applied on selected join points. LARA borrows several constructs from the JavaScript language, including arrays, loops, and function calls. This allows complex strategies to be written, including DSE strategies. A *condition* specifies whether an action should be applied to a join point. LARA aspects can specify transformation and mapping recipes capturing a wide range of implementation schemes to satisfy NFRs.

Figure 2 shows an example of the weaving process triggered by the aspect definition below. In this example, we specify a pointcut expression (*select* section) that selects all `for`-loops in the C source. A pointcut expression defines a chain of elements according to the hierarchy of the C syntax structure. The result of the pointcut expression is a table, where each row corresponds to a chain of join points. The weaving process traverses each row in the table, checks the *condition* associated to the apply *section*, and performs the corresponding actions. The values of join point attributes, such as loop type or number of iterations, can be accessed inside conditions and actions as the weaving process traverses the chain of join points. By default, actions are applied to the last join point in the chain (in our example, *$loop*), but any chain element can be specified as a target by using its reference (e.g., in the *apply* below, *$function* corresponds to all functions in the code with at least one `for`-loop). Note that the LARA front-end allows the specification of join point, action and attribute models. Hence, LARA can target different languages and systems by revising these models.

```
aspectdef loopunroll
   select function.loop{type==”for”} end
   apply optimize(“loopunroll”, “fully”); end // fully unroll loop
   condition $loop.is_innermost && $loop.num_iter<=32; end
end
```


original
Select
function.loop{type=="for"}
join point table
after weaving
attributes for $loop
$loop.is_innermost && $loop.num_iter <= 32
true
apply optimize loopunroll; end
Apply
Condition

```
for(i=0; i<10; i++) {
  for(j=0; j<15;j++) {
    …
  }
}
```

function	loop
main()	29:1 - for(…) { }
main()	30:3 – for (…) { }
foo()	40:6 – for (…) { }

```
for(i=0; i<10; i++) {
  /*loop j fully
    unrolled) */
}
```

type	is_innermost	num_iter
for	true	15

Figure 2. Weaving process in LARA.

In addition to a pre-defined set of attributes associated to each join point, new attributes can be defined by developers. For instance, profiling information generated by tools like *gprof* can be annotated back as attributes in LARA (see top of Figure 1) as illustrated in the aspect below. In this example, we added attributes *time* and *calls* to functions *f1* and *f2*, which can be exported to other aspect definitions.

```
aspectdef gprof_results
   select function{name==”f1”} end
   apply $function.time = 30; $function.calls = 15; end
   select function{name==”f2”} end
   apply $function.time = 54; $function.calls = 32; end
end
```

Next, we briefly describe the target architecture used for the experiments and implementations presented herein.

C. *Target Architecture*

For the hardware/software solutions presented in this paper we target FPGA implementations with a GPP coupled to hardware accelerators, as illustrated in Figure 3. The implementations use a PowerPC hardcore (PPC440) as GPP, which executes the application software components, and hardware modules, coupled to the PLB (processor local bus), responsible for the hardware components. The PPC core is connected to a single-precision FPU (floating-point unit) via the APU (auxiliary processor unit) interface. This base configurable architecture has been implemented on a Virtex-5 FX FPGA with a PPC clocked at 400 MHz and the CCUs at 100 MHz.

The hardware modules consist of a number of CCUs (custom computing units) and local memories. The CCUs do not have direct access to the non-local memories (e.g., external memories) and data communications to and from the hardware modules are performed by the PPC. In particular, data communication is done by transferring via the PPC the data from external memories to hardware local memories and vice versa. This architecture allows the PPC and the CCUs to execute concurrently. The execution of the CCUs is controlled by the PPC via *start* and *done* signals.

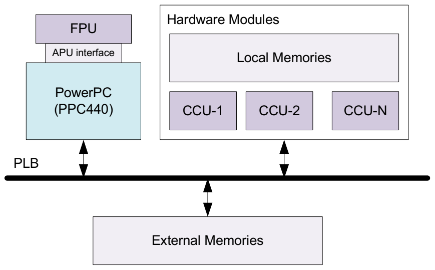


Figure 3. Block diagram of the target architecture.

III. Case Studies

To evaluate the impact of a diverse set of optimization strategies on the global application speedup, we selected two applications [5] - 3D Path Planning (3DPP) and Stereo Navigation (SN) - provided in the context of the REFLECT project [6]. Both applications are specified in plain C code and their execution time is concentrated in a reduced number of functions selected for hardware acceleration. Table I shows the main characteristics of the two applications using common software quality metrics. 3DPP and SN have 841 and 9,027 lines of code spread, respectively, over 48 and 164 functions. SN has a higher cyclomatic complexity than 3DPP revealing that SN is more control intensive.

Table I. Software metrics for the two applications.

Metric	3D-Path-Planning (3DPP)	Stereo Navigation (SN)
Blank/Code/Comment Lines	186/841/107	1,427/9,027/2,490
Lines/ Files/ Functions	1,241/10/48	13,730/68/164
Declarative/Executable Statements	273/505	1,798/5,647
AvgCyclomatic/ MaxCyclomatic	3.1/12	6.65/72
MaxNesting/ CountPath	7/314	6/4,022,376,475
SumCyclomatic/ SumEssential	149/48	1,091/193

3DPP computes in real-time a possible path to be followed by an aerial vehicle flying across an environment with known obstacles. Profiling analysis identified the function *gridit* as responsible for about 90% of the global execution time. Figure 4 presents the main computations of *gridit*. This function is invoked in different points in the program, with 3 different sizes (referred herein as: full, div2, and div4) of the 3D matrices *pot* and *obstacles*. The innermost loop updates each element of the matrix *pot*, either assigning to a constant value or to the average of the 6 neighbors of the element being updated (see Figure 4).

SN combines two images from two cameras to identify the relative movement of image features, thus providing vehicle navigation data, such as its absolute translation and rotation. The software profiling of SN led to the identification of the *HarrisTile* stage, which corresponds to a Harris corner detector and is responsible for more than 60% of the global execution time. This stage consists of convolution operations using three functions: *ConvC*, *ConvR1*, and *ConvR2*. All three functions work on a 96×96 2D single-precision floating-point array, with a multiply-accumulate operation at their innermost loops.

```
#define ITER_STEPS_NUM …
void gridit(int* obstacles, float* pot)  { …
  for (it = 0; it < ITER_STEPS_NUM; it++) {  // loop it
    for (i = 1; i < (X_DIM - 1); i++) { // loop x
      for (j = 1; j < (Y_DIM - 1); j++) { // loop y
        for (k = 1; k < (Z_DIM - 1); k++) { // loop z
          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 = pot[i-1][j][k]+pot[i+1][j][k]+pot[i][j-1][k]+
                   pot[i][j+1][k]+pot[i][j][k-1]+pot[i][j][k+1];
                   pot[i][j][k] = FIX(acc * SCALE);  }}}}}
```

Figure 4. Simplified code for *gridit* (highlighted statements result from code transformations).

IV. Compiler Strategies

We now describe the key transformations and mapping choices associated with each of the two applications, namely 3D Path Planning (3DPP) and Stereo Navigation (SN).

The 3DPP strategies described here target *gridit* and the code surrounding its invocations. Table II briefly describes the considered transformations and optimizations. We use them to define specific strategies such as the ones specifically applied to *gridit* (see Table III) and to 3DPP (see Table IV). An optimization example is *loop fission and move* which can be applied to the *it* loop, a loop with 12 iterations located at the main 3DPP function. Nested within this loop are 5 invocations of *gridit*. Moving this loop to *gridit* (see Figure 4) results in 5 invocations of *gridit* per algorithm step rather than the 60 invocations originally required. This reduction will have a high impact in the global HW/SW implementation (see Section V).

Table II. Transformations and optimizations for 3DPP.

Name	Description
Loop fission and move	Move *iter* loop to *gridit* function body
Replicate array k×	Replicate k× array *pot* and map each array in a distinct on-chip memory
Map *gridit* to HW core	Synthesize the *gridit* function with Catapult-C
Specialization	Each call statement to *gridit* transformed to a call to a specialized function according to the size of the map (three possible sizes used: full, half, quarter)
Pointer-based accesses and strength reduction	Data transfer to HW using pointers and pointer-arithmetic (additions) instead of using array accesses which require more complex structures for address calculations (e.g., multiplications by 4 of an index and addition to the base address).
Unroll k×	Unroll k times the innermost loop of *gridit*
Eliminate array accesses (EAS)	Eliminating repetitive loads by using a scalar to store the first load
Move data access	Move data communication from the local HW core memories to the code location where data are needed
Loop coalescing-k	Considers coalescing of the first k inner loops in *gridit* function.
Transfer data according to *gridit* call	Transfer the arrays (*obstacles* and/or *pot*) according to the size needed for a given *gridit* call.
On-demand *obstacles* data transfer	Transfer array *obstacles* only when needed and promoting the reuse between hardware core executions of its storage in local memories.
Moving array accesses (MAA)	Move accesses (read/write) to array *pot* to outside the if-else construct.

The strategies in Table III and Table IV are codified using the LARA language. Figure 5 shows some examples of aspects for specifying individual optimizations such as array replication, loop unrolling, loop coalescing, arrays to memory mapping, and an aspect defining a strategy (in this case the strategy 4b presented in Table III). Note that in this example we replicate array *pot* to allow the generated hardware core exploit concurrent load/store operations.

The use of input parameters in the first four aspects in Figure 5 makes those aspects reusable and thus possible to apply to other functions besides *gridit*.

Table III. Compiler strategies applied to *gridit*. x, y, z represent the loops involved.

Strategy	Sequences of application
1b	Repl. pot 3×
1c	Repl. pot 6×
2a	Unroll 2× loop z → EAS
2b	Repl. pot 3× → Unroll 2× loop z → EAS
2c	Repl. pot 6× → Unroll 2× loop z → EAS
3a	coalesce loops y,z
3b	Repl. pot 3× → coalesce loops y,z
3c	Repl. pot 6× → coalesce loops y,z
4a	Unroll 2× loop z → EAS → coalesce loops y,z
4b	Repl. pot 3× → Unroll 2× loop z → EAS → coalesce loops y,z
4c	Repl. pot 6× → Unroll 2× loop z → EAS → coalesce loops y,z
5a	Unroll 4× loop z → EAS
5b	Repl. pot 3× → Unroll 4× loop z → EAS
5c	Repl. pot 6× → Unroll 4× loop z → EAS
6a	coalesce loops x,y,z
6b	Repl. pot 3× → coalesce loops x,y,z
6c	Repl. pot 6× → coalesce loops x,y,z
7a	coalesce loops x,y,z→ Unroll 2× loop z → EAS
7b	Repl. pot 3× → coalesce loops x,y,z→ Unroll 2× loop z → EAS
7c	Repl. pot 6× → coalesce loops x,y,z→ Unroll 2× loop z → EAS
8a	Unroll 2× loop z → coal. loops y,z → EAS → MAA
8b	Repl. pot 3×→ Unroll 2× loop z → coal. loops y,z → EAS → MAA
8c	Repl. pot 6×→ Unroll 2× loop z → coal. loops y,z → EAS → MAA

Table IV. Strategies for 3DPP using Table II optimizations.

Optimization	Strategy							
	1	2	3	4	5	6	7	8
Loop fission and move		✓	✓	✓	✓	✓	✓	✓
Replicate array 3×					✓	✓	✓	✓
Map *gridit* to HW core	✓	✓	✓	✓	✓	✓	✓	✓
Pointer-based accesses and strength reduction			✓	✓	✓	✓	✓	✓
Unroll 2×	✓	✓	✓	✓	✓	✓	✓	✓
Eliminating array accesses	✓	✓	✓	✓	✓	✓	✓	✓
Move data access								✓
Specialization → 3 HW cores							✓	✓
Tranfer *pot* data according to *gridit* call				✓		✓	✓	✓
Tranfer *obstacles* data according to *gridit* call			✓	✓	✓	✓	✓	✓
On-demand *obstacles* data transfer			✓	✓	✓	✓	✓	✓

With respect to SN, as the Catapult-C version used in the experiments does not directly support floating-point data types, we modified the source code to use software implementations of the floating-point multiplication and addition operators based on the *softfloat* library [7], but manually simplified by eliminating the treatment of special numbers, such as NaN. Even though this approach does not map the operators into highly optimized arithmetic cores, it allows Catapult-C to synthesize floating-point arithmetic to hardware with interesting results.

Table V presents the optimizations applied to SN. They consider the parallel execution of CCUs, the implementation of floating-point operations using *softfloats*, the allocation of certain arrays to local memories, and the transformation of floating-point data to integer data whenever possible.

```
aspectdef id2
 input k, name1=“gridit”, name2=“pot” end
 select function{name1}.var{name2} end
 apply optimize(“replicate”, “array”, k); end
end
aspectdef map2mem
 input name1=“gridit”, name2=“pot” end
 var id = 0;
 select function{name1}.var{name2} end
 apply map(to:“BRAM”, id:id++, array:$var.name); end
end
aspectdef id6
 input k, level, name1=“gridit” end
 select function{name1}.loop end
 apply optimize(“loopunroll”, k);end
 condition $loop.numIterIsConstant &&
     $loop.is_innermost && $loop.level = level end
end
aspectdef id10
 input level1, level2, name1=“gridit” end
 S1: select function{name1}.($l1=loop) end
 S2: select function{name1}.($l2=loop) end
 apply to S1::S2 optimize(“loopcoal”);end
 condition $l1.level==level1 && $l2.level==level2 end
end
aspectdef Strategy_4b
 call id6(2, 3); optimize(“scalarreplace”);
 call id10(2, 3); call id2(3); call map2mem();
end
```

Figure 5. Examples of aspects for individual optimizations and an aspect specifying a strategy (bottom).

Table V. Transformations and optimizations for SN.

Name	Description
Softfloats	Transformation of floating-point operations and data types to *softfloat* implementations
Parallel execution of CCUs	The HW cores of two convolutions executed in parallel.
Local promotion	Arrays of constants (input) passed as parameters promoted to local arrays
Float to integer	Transformation of single precision floating-point data types to 32-bit integers

V. Experimental Results

This section presents experimental results for the 3D Path Planning (3DPP) and Stereo Navigation (SN) applications described in Section III. For each of these applications we report the results using the compilation and synthesis strategies described in the previous section. Note that although previous work [8] preliminarily illustrated the impact of some strategies when applied to isolated functions, in this section we focus on the impact on the entire application execution instead.

The software/hardware solutions presented here were implemented in an ML507 board, which includes a Xilinx Virtex-5 (XC5VFX70TFFG1136) FPGA. The system architecture was designed using Xilinx EDK12.3, the hardware cores were generated using Catapult-C [3] and their RTL-Verilog descriptions were synthesized using the Mentor Graphics Precision Synthesis tool (version 2010a 2.254). For the backend stages we used the tools included in Xilinx ISE 12.3.

The target architecture (see Figure 3) considers the PPC hardcore found in the Virtex-5 FPGA, the external memory for data and program instructions, and hardware cores connected to the local bus. For all the implementations, the PPC runs at 400 MHz and the software code was compiled with *ppc-gcc* with flag –O2. We use as a starting reference the embedded software implementations of the applications.

The designs described herein differ as the result of the sequence of source code transformations that were the target of, or in the way the corresponding codes were generated and/or mapped to the available hardware resources.

A. 3D Path Planning (3DPP)

The 3DPP application (described in Section III) uses two 3D arrays for the *obstacles* map and for the *potential* field respectively. All the results presented in our evaluation use a map with the size 32×64×16. The various implementations have variations on the strategies to map *gridit* (the hotspot function in 3DPP) to hardware and to communicate data between the PPC and the hardware cores. For all 3DPP designs we use fixed-point data types.

Figure 6 presents the impact of all strategies (optimizations) listed in Table III regarding the resource usage and speedup achieved by each hardware implementation compared to the *gridit* function without optimizations (named as 1a). A speedup of 2.33× was achieved with strategy 5c (see Table III), but the corresponding design requires 2.43× more resources than the baseline implementation. Strategy 2c led to a design that achieves a speedup of 2.15× and requires only 1.68× more resources, thus providing a better trade-off between performance and resource utilization. Replicating the *pot* array 3× instead of the 6×, as used for designs 5c and 2c, attains typically a similar performance using more slices and less BRAMs. Strategy 2b led to a speedup of 1.91× requiring only 1.79× more resources. While avoiding array replications, strategy 8a leads to a speedup of 1.40× requiring 1.62× more resources.

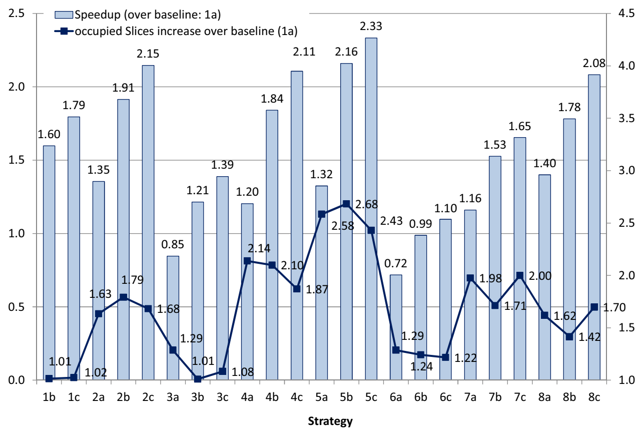


Figure 6. Impact of different strategies on the speedup and hardware resources for the *gridit* function.

With respect to the implementation of the entire 3DPP application, Table VI presents the resources for the hardware module (see Figure 3) considering the 8 HW/SW implementations (Table IV). The implementations using 3 specialized cores (7 and 8) clearly require more FPGA resources. Note that we do not consider for the entire HW/SW implementations the hardware cores of *gridit* with the *pot* array replicated 6× as it is a version that in practice hardly justifies the large number of BRAMs.

The maximum speedups obtained by the hardware cores used over the software implementation were 12.15× and 27.30× with and without communication costs, respectively.

Table VI. FPGA resources used for the hardware cores in each 3DPP implementation.

FPGA resources	Implementation			
	1	2,3,4	5,6	7,8
# Slice Registers used as Flip Flops	901	939	956	2,470
# Slice LUTs	1,182	1,284	1,308	2,148
# occupied Slices	531	663	642	1,004
# BlockRAM/# DSP48Es	34/6	34/6	98/6	98/12

Lastly, we show in Figure 7 the overall speedups achieved considering each of the 8 implementations in Table IV. The significant increase in the speedup between implementations 1 and 2 results from the application of *loop fission and move* (see Table IV). This led to a reduction of execution calls to the HW core, and thus to lower communication costs. Also, the moved loop is now implemented by the HW core. Although implementations 7 and 8 use the HW cores running at 85 MHz instead of 100 MHz (this decrease in clock frequency occurs when adding these cores to the entire system, and requires higher P&R effort), this does not prevent these implementations to achieve the highest speedups. In fact, implementation 8 attains an overall speedup of 6.8× over the software version. This is significant as the code optimizations and transformations do not rely on a deep restructure of the algorithm.

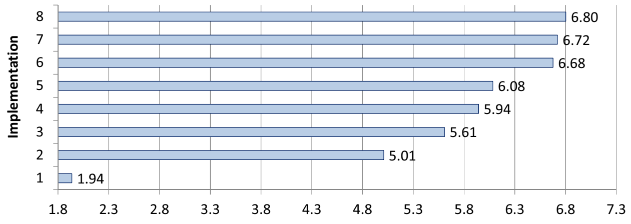


Figure 7. Overall speedups for different implementations of 3DPP using Table IV strategies.

B. Stereo Navigation (SN)

SN (described in Section III) uses single precision floating-point data types. All the hardware cores were generated using the *softfloat* versions of the floating-point operations. The final SN implementation uses three hardware cores, one for the convolution *ConvC* and the other two for convolutions *ConvR1* and *ConvR2* of the *HarrisTile* stage of the application. This implementation takes advantage of the parallel execution of the three hardware cores staging the computations in 5 stages, three of them executing two convolutions concurrently (of the 8 convolutions in the *HarrisTile* stage). This solution uses 5 local memories with size 16384×32-bit for storing the arrays, 4 local memories with size 16×32-bit, and 2 local memories with size 8×32-bit for storing the constants of the *ConvC* convolutions.

Figure 8 presents the speedups achieved. The hardware cores used for convolutions allowed *HarrisTile* implementations with speedups of 6.62× and 3.90× over pure software implementations, when compiling both versions with –O0 and –O2, respectively. In addition, the entire HW/SW implementation of SN achieves global speedups of 2.12× and 2.53× over the software implementations with –O2 and –O0, respectively. The use of the strategy considering the concurrent execution of two convolutions is responsible for about 10% overall improvement of the speedup.

Table VII presents the FPGA resources used for each of the three hardware cores according to the optimizations presented in Table V. The complexity of the cores responsible for *ConvR1* and *ConvR2* is similar while the core for *ConvC* occupies much less FPGA resources due to the transformation of floating-point to integer representations. All three hardware cores were clocked at 100 MHz.

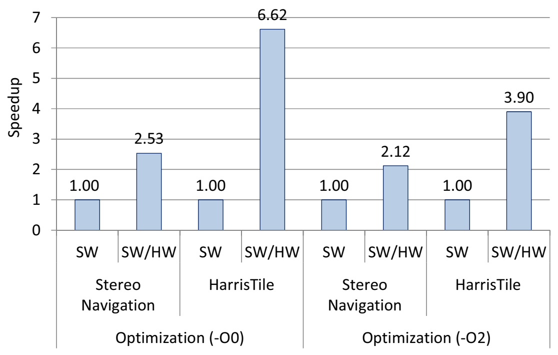


Figure 8. Speedups achieved for the entire application (Stereo Navigation) and for the *HarrisTile* stage.

Table VII. FPGA resources used for the hardware cores.

FPGA resources	hardware core		
	ConvC	ConvR1	ConvR2
# Slice Registers used as Flip Flops	1,032	5,987	5,899
# slice LUTs	1,180	11,676	11,709
# occupied Slices/# DSP48Es	726/3	4,492/7	4,304/7
# BlockRAM 36k / # BlockRAM 18k			6/160

VI. RELATED WORK

This section briefly describes the related work concerning compiler optimizations as well as relevant aspect-oriented programming (AOP) approaches.

A. Compiler Optimizations

In contexts other than AOP, researchers have developed compilation systems for performance tuning. While earlier efforts focused on empirical-based approaches (e.g., ATLAS [9]), where programmers would simple let their applications run and use the output of metrics to decide which sequence of transformations were the best, later efforts addressed more systematic approaches, and used a combination of performance models (e.g., [10]) and special purpose (or domain-specific) languages for defining compiler transformations sequences (e.g., [11]). Other approaches allow developers to customize the composition and parameterization of transformations through scripting, in order to automatically derive implementations that can meet the specified goals [12]. In the context of multi-core platforms, there were efforts focused on DSE approaches, such as the one presented in [13], for generating multi-objective optimizing strategies based on available performance metrics and constraints.

In the context of synthesizing high-level programming languages such as C to FPGAs, many efforts also considered a wide range of program transformations. Given the complexity of the target reconfigurable architectures, these efforts leveraged a series of algorithmic techniques to mitigate the huge design spaces to explore. Rather than dealing with long synthesis times, researchers have developed estimation-based approaches where the effects of specific transformations on the resulting designs were modeled as described, e.g., in [14]. The hArtes toolchain [4] maps C applications to multi-core platforms by automatically partitioning the application and mapping the most suited tasks to accelerators such as FPGAs. The partitioning and mapping process are based on heuristics and profiling, and are guided by user-defined source annotations.

We believe that LARA can complement the above approaches by providing a unifying framework, capable of capturing and enacting evolving strategies with full control over the design-flow. For instance, LARA is currently being used to control via aspects design choices such as hardware/software partitioning. In general, the use of aspects differs from these approaches in various subtle ways. From a transformational perspective, LARA presents a simple paradigm of select/transform rather than an imperative approach to apply a strict sequence of transformations. This allows LARA to be more flexible and composable than other more monolithic approaches [15]. Second, it allows developers to convey to transformation engines key metrics of performance in a simple and unified form. In various other systems these metrics are typically hidden from the developer and only indirectly observed in other metrics of the resulting design, making the overall navigation of the design space extremely difficult if not impossible.

B. Aspect-Oriented Programming

The LARA design has been influenced by other AOP languages [2] such as AspectJ [16] and AspectC++ [17]. In our approach, pointcut expressions define composable select expressions (similar to composable queries) as in [18]. 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, 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 NFRs of the target design solutions. Based on these requirements, LARA supports not only actions associated with the join point's code to be executed (as in AspectJ), but also compiler optimization directives and data and type information about variables.

In addition, the separation of concerns in LARA 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 [19]). 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 design portability and retargetability. By codifying concerns, such as performance as aspects our approach 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.* [20] in the context of software product lines. In addition, by exposing the characteristics of the target architecture to aspects, 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.

VII. Conclusions

This paper described the use of a novel aspect-oriented hardware/software design-flow for FPGA-based embedded platforms. The design-flow uses LARA, a domain-specific aspect-oriented programming language that supports the high-level specification of compilation and application mapping strategies, including sequences of data/computation transformations and optimizations. We illustrated the use of LARA on two complex real-life applications using high-level compilation and synthesis strategies achieving complete hardware/software implementations with speedups of 2.5× and 6.8× over software-only implementations. Our experiences indicate that our AOP approach enhances design reuse while preserving the original application source-code, thus promoting developer productivity as well as architecture and performance portability.

Ongoing work is addressing the integration of optimizations and code transformations in the weaving phases of the toolchain. Specifically, we are extending the support and the specification of LARA strategies in the context of design space exploration.

Acknowledgment

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.

References

[1] J. M. P. Cardoso, *et al.*, "LARA: An Aspect-Oriented Programming Language for Embedded Systems," in *Int'l Conf. on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany, March 25-30, 2012.

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

[3] Mentor Graphics Corp. *Catapult C Synthesis C to Hardware Concepts*. October 2009.

[4] W. Luk, *et al.*, "A High-Level Compilation Toolchain for Heterogeneous Systems," in *Proc. IEEE Int'l SOC Conf. (SOCC'09)*, Belfast, Northern Ireland, Sept. 9-11, 2009, pp. 9-18.

[5] ICT-2009-4 REFLECT, April 2009. Deliverable 1.1 "Repository of applications from Honeywell".

[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. Hauser, June 2011. Available at http://www.jhauser.us/arithmetic/SoftFloat.html.

[8] 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.

[9] R. Whaley, and J. Dongarra, "Automatically Tuned Linear Algebra Software," In *Proc. of ACM Supercomputing Conference (SC'98)*, 1998, IEEE Computer Society, Washington, DC, USA, pp. 1-27.

[10] A. Tiwari, C. Chen, J, Chame, M. Hall, and J. Hollingsworth, "A Scalable Auto-Tuning Framework for Compiler Optimizations," In *Proc. Int'l Symp. on Parallel and Distributed Processing (IPDPS'09)*, 2009, IEEE Comp. Society, Washington, DC, pp. 1-12.

[11] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A Language and Compiler for DSP Algorithms," In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'01)*, 2001, ACM, New York, NY, USA, pp. 298-308.

[12] Q. Liu, T. Todman, J. Coutinho, and W. Luk, G. Contantinides "Optimising designs by combining model-based and pattern-based transformations," In *Proc. Int'l Conf. on Field Programmable Logic and Applications (FPL'09)*, Prague, Czech Republic, Aug. 31-Sept. 2, 2009, pp 308-313.

[13] G. Palermo, C. Silvano, and V. Zaccaria, "Multi-Objective Design Space Exploration of Embedded Systems," *in Journal of Embedded Computing*, 2005, Vol. 1, No. 3, pp. 305-316.

[14] J. Gerlach, W. Rosenstiel, and B. Gregory, "A Methodology and Tool for Transformation-based High Level Design Space Exploration," In *Proc. Int'l Conf. on Computer Design (ICCD'00)*, Austin, TX, USA, 2000, pp. 454-548.

[15] M. Voss, and R. Eigenmann, "High-Level Adaptive Program Optimization with ADAPT," In *Proc. ACM Symp. on Principles and Practices of Parallel Programming (PPoPP'01)*, 2001, ACM, New York, NY, USA, pp. 93-102.

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

[17] O. Spinczyk, A. Gal, and 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 (CRPIT'02)*, Australian Computer Society, Inc., Darlinghurst, Australia, 2002, pp. 53-60.

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

[19] R. Ferrer, *et al.*, "Optimizing the Exploitation of Multicore Processors and GPUs with OpenMP and OpenCL," in *Proc. 23rd Int'l Conf. on Languages and Compilers for Parallel Computing (LCPC'10)*, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 215-229.

[20] 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.), LNCS, Vol. 4640. Springer-Verlag, Berlin, Heidelberg, 2007, pp. 117-142.