SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **46**:251–287
Published online 11 December 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2301

Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach

João M. P. Cardoso[1,*,†], José G. F. Coutinho[2], Tiago Carvalho[1], Pedro C. Diniz[3], Zlatko Petrov[4], Wayne Luk[2] and Fernando Gonçalves[5]

[1]*Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal*
[2]*Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2BZ London, UK*
[3]*Information Sciences Institute, USC, Marina del Rey, Los Angeles, CA 90089-0911, USA*
[4]*Honeywell International s.r.o, Turanka, 100 627 00 Brno, Czech Republic*
[5]*Coreworks S.A., Rua Alves Redol 9, 1000-029 Lisboa, Portugal*

SUMMARY

The development of applications for high-performance embedded systems is a long and error-prone process because in addition to the required functionality, developers must consider various and often conflicting nonfunctional requirements such as performance and/or energy efficiency. The complexity of this process is further exacerbated by the multitude of target architectures and mapping tools. This article describes LARA, an aspect-oriented programming language that allows programmers to convey domain-specific knowledge and nonfunctional requirements to a toolchain composed of source-to-source transformers, compiler optimizers, and mapping/synthesis tools. LARA is sufficiently flexible to target different tools and host languages while also allowing the specification of compilation strategies to enable efficient generation of software code and hardware cores (using hardware description languages) for hybrid target architectures – a unique feature to the best of our knowledge not found in any other aspect-oriented programming language. A key feature of LARA is its ability to deal with different models of join points, actions, and attributes. In this article, we describe the LARA approach and evaluate its impact on code instrumentation and analysis and on selecting critical code sections to be migrated to hardware accelerators for two embedded applications from industry. Copyright © 2014 John Wiley & Sons, Ltd.

Received 20 December 2013; Revised 15 October 2014; Accepted 29 October 2014

KEY WORDS: aspect-oriented programming; compilers; hardware/software systems; embedded systems; domain-specific languages; monitoring; instrumenting and profiling

1. INTRODUCTION

The development and mapping of applications to contemporary heterogeneous and possibly multicore, high-performance embedded systems require tools with very sophisticated design flows. Developers must be aware of critical applications requirements, both functional and nonfunctional (e.g., real-time performance and safety) while meeting their 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 analyze the application code (or another form of functional specification) and partition it among the most suitable system components through a process commonly known as hardware/software partitioning [1]. Subsequently, they have to deal with multiple compilation tools that target each specific system

*Correspondence to: João M. P. Cardoso, Departamento de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal.
†E-mail: jmpc@acm.org

component. These problems are exacerbated when dealing with field-programmable gate arrays (FPGAs), a popular hardware technology for fast prototyping that combines the performance of custom hardware with the flexibility of software [1, 2]. As a consequence, developers must explore code and mapping transformations specific to each architecture so that the resulting solutions meet overall requirements. As the complexity of emerging heterogeneous embedded systems continues to increase, the need to meet increasingly challenging implementation trade-offs (e.g., energy efficiency and performance) will undoubtedly increase the complexity of mapping sophisticated applications to these embedded systems. Another issue contributing to development complexity lies in the existence of different product lines for the same application, a characteristic increasingly present given the need to support multiple target platforms.

One of the key stages of the mapping process is to profile the code for the purpose of understanding its performance behavior, which is commonly achieved by extensive code instrumentation and monitoring. Code instrumentation is becoming increasingly important in the context of profiling and monitoring to support: the analysis of application behavior, the identification of regions for parallelization and/or mapping to accelerators, and the assessment of the impact of specific optimizations and design decisions (e.g., Refs. [3–5]). Approaches providing flexible and programmable mechanisms for extracting runtime information are of paramount importance given the increasing role of customization. The current *de facto* standard practices for the development of applications targeting high-performance embedded systems relies extensively on architecture-specific source code transformations and/or on the use of tool-specific compilation directives. Such practices lead to low developer productivity and to limited application portability, as when the underlying architecture changes developers invariably need to restart the design process.

To address these challenges, we designed the approach inspired by aspect-oriented programming (AOP) [6, 7], conceptually illustrated in Figure 1. An important element of this approach is LARA, an AOP language (a first version was introduced in Ref. [8]) to support developers in mapping their

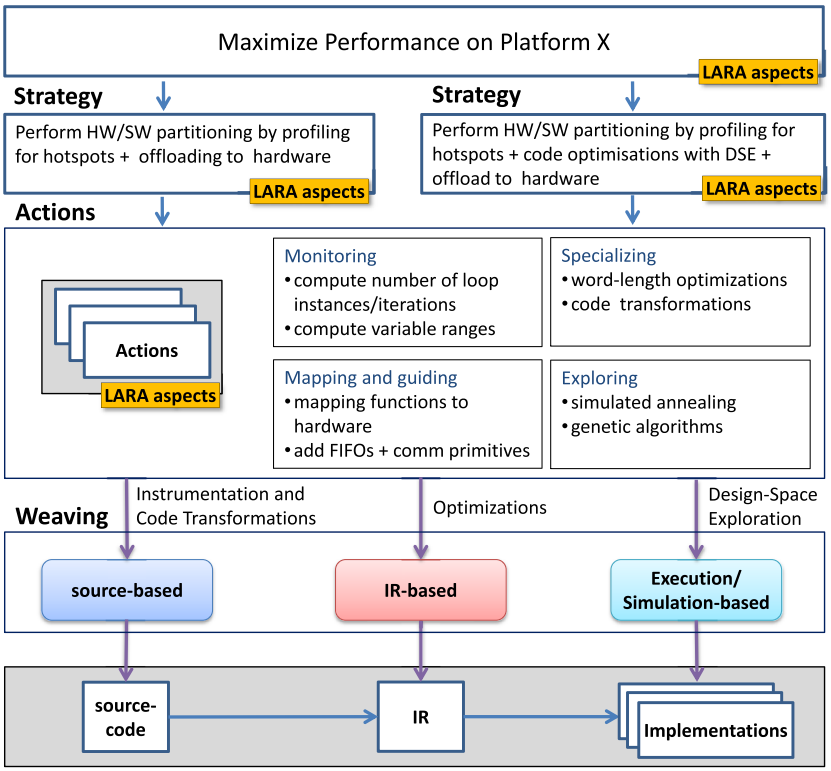


Figure 1. Overview of the LARA approach.

DOI: 10.1002/spe

applications to embedded systems (including heterogeneous high-performance systems) across various design stages including code analysis, monitoring, mapping decisions, and hardware/software compiler optimizations. In addition to supporting traditional AOP mechanisms, our approach focuses on mechanisms that address nonfunctional requirements (NFRs), including crosscutting concerns exposed by various design elements, such as the application source code, the tools used in the design flow, and the target platforms.

LARA allows developers to capture application NFRs in a structured way, leveraging high/low-level actions and flexible toolchain interfaces. Developers can thus benefit from retaining the original application source code while also leveraging the benefits of an automatic approach for various domain-specific and target component-specific compilation/synthesis tools. Specifically, LARA has been designed to help developers reach efficient implementations with low programming effort. In essence, LARA uses AOP mechanisms to offer in a unified framework: (i) a vehicle for conveying application-specific requirements that cannot otherwise be specified in the original programming language; (ii) the use of requirements to guide transformations and mapping choices, thus facilitating design space exploration (DSE); and (iii) an extensible interface for the various compilation/synthesis components of the toolchain.

As illustrated in Figure 1, LARA aspects are used to describe concerns (possibly crosscutting) using different abstraction levels, from the specification of the concerns to strategies based on actions to implement those concerns. This allows a unified view of concerns and the use of the same language (avoiding the need to learn more than one language) over different specific concretizations and design flow levels. Figure 1 also illustrates the different levels of abstraction using a specific crosscutting concern: 'maximize performance on platform X'. This concern is specified and applied as LARA strategies. These LARA strategies may consist of monitoring, mapping and guiding, specializing, and DSE actions. The actions are then applied in a weaving stage that can be at the source-to-source level, at the compiler optimizations level, and/or at the DSE level. The work described here focuses on the use of concerns and strategies implemented using a source-to-source weaving approach as depicted in Figure 1.

This article makes the following specific contributions:

- Describes the syntax and the semantics of LARA, an AOP language capable of capturing transversal (across multiple modules and/or applications) and vertical concerns (across different stages of a design flow).
- Describes the LARA join point model and the flexible and extensible join point metaproperty attributes, which allow LARA strategies to use information from application profiling and analysis.
- Evaluates the LARA approach through the use of complex applications and considering a number of concerns with significant importance in the context of developing embedded applications.
- Presents experimental results of the use of LARA for a set of real-life applications targeting a heterogeneous embedded architecture mainly consisting of a general-purpose processor (GPP) and a reconfigurable coprocessor.

This article describes the research and development of LARA mainly in the context of the REFLECT research project [9]. The selected case study and associated requirements were provided by an industrial partner of the REFLECT consortium.

This article is organized as follows. Section 2 describes the main motivation and the rationale for the work presented in this paper. Section 3 provides an overview of our aspect-oriented approach and design flow. Section 4 focuses on the LARA language with a number of illustrative examples. Section 5 shows the practical impact of our approach with a number of benchmarks and with industrial applications. In Section 6, we present the use of our approach in the context of two real-life industrial applications targeting an embedded computing system consisting of a GPP and a customizable coprocessor. In Section 7, we describe related work and conclude in Section 8.

2. MOTIVATION

The development of high-performance embedded applications is a notoriously complex process as developers need to master various design tasks, toolchains, and target architecture details. The complexity rises as embedded systems typically consist of a myriad of architectures with various

DOI: 10.1002/spe

heterogeneous components, which requires a development process that takes into account hardware/software partitioning and mapping. In addition, the complexity of mapping applications expressed in high-level imperative programming languages to heterogeneous reconfigurable computing architectures (as is the case of FPGA-based architectures) is further exacerbated by the variety of mapping tools and computing paradigms these architectures expose. Typical design flows include tools that provide distinct interfaces and are controlled by a wide range of parameters, thus creating a large and complex design space with distinct controlling features.

2.1. Requirements

Developers have to deal with a set of strict NFRs and face a number of tasks, which include carrying out a detailed performance analysis study for selecting the set of computational kernels that can benefit from source-level code transformations and/or hardware acceleration. NFRs may include performance, power/energy restrictions, real-time constraints, reliability, safety, computer architecture, and device restrictions. The best solution for specific requirements, such as input data rates, may simply not be feasible considering other requirements, such as the ones imposed by constraints such as size or power. Strict NFRs, such as reliability, safety, performance, and energy consumption, are usually not supported by existing tools or cannot be easily expressed using current programming languages.

As an example, let us consider mapping a C implementation of an MPEG-2 audio encoder (layers I and II) to an embedded system consisting of a GPP coupled to reconfigurable hardware on which we map custom accelerators for executing critical sections of the application. In this context, Table I presents a list of concerns identified by our industry partner. These concerns span the application development cycle from the analysis phase (with the use of instrumentation to monitor specific properties) to the mapping to the target architecture. More specifically, the different stages of a development cycle typically include monitoring, fine-tune optimizations, hardware/software partitioning, and synthesis/compilation. Most of these concerns are crosscutting and are common to other applications.

2.2. Challenges to address requirements

The development process for embedded systems usually includes the following tasks: instrumentation of the source code for monitoring and custom execution profiling; code transformations and

Table I. Examples of concerns applied to an MPEG audio encoder application source code.

Type of aspects	Examples
Monitoring (C code insertion)	Monitor range of the variables **z** and **m** in function **fsubband** for word length optimization After word length optimization, monitor variables **s** and **y** to validate deviations from their original values Replicate the **fsubband** function body with different word lengths to do deviation analysis 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** and monitor the possible deviations introduced
Mapping and guiding	Map functions **add_sub**, **fsubband**, and **II_f_f_t** to the hardware accelerator 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 key hardware blocks in any audio system. In a specialized version, these hardware blocks need to interface with the **read_samples** function. Generate from the unmodified original code a code version interfacing to these hardware blocks Bind the functions **pow** and **log10** in the **II_f_f_t** and **add_db** functions to hardware Map arrays in functions **fsubband** and **II_f_f_t** to external memories

DOI: 10.1002/spe

optimizations regarding, for example, performance; hardware/software partitioning; and exploration of various design alternatives. In most cases, these tasks are performed with independent tools and approaches, without an integrated view. For instance, the results of the profiling are manually used to identify critical functions. Furthermore, the application source code is usually instrumented to expose information that can be used in various stages of the design flow, to help analyze and identify critical sections, data dependences, and specialization opportunities in support of optimization goals that include performance, resource utilization, energy consumption, and safety.

In order to satisfy NFRs, one may need to apply a set of strategies, possibly taking into account best practices. These strategies usually imply a sequence of tasks, such as a set of compiler optimizations and code transformations, the use of specific fault-tolerance schemes, and customizing the execution of the toolchain. Therefore, we consider important to have an approach that allows developers to express such strategies in a systematic way, supported by tools that can apply them automatically, and providing a single and unified view of the entire strategy. In particular, this unified view should cover all stages of the design flow, including profiling and mapping.

2.3. Using aspects to address requirements and challenges

The rationale for the choice of an AOP approach is backed up by the wide experiences of using LARA aspects for software/hardware transformations [9], which have revealed the benefits of AOP in application code development, including program portability across architectures and tools, and productivity improvement for developers. Next, we describe how our aspect-oriented approach can address these concerns by providing an example of how LARA can enhance the development process.

We express the concerns as LARA aspects and use our weaving tools to automatically apply these aspects to the application source code of an MPEG encoder. In a first stage, we identify the critical functions corresponding to possible performance bottlenecks by profiling the application on a PC environment. Using LARA aspects and our tools, we then automatically instrument these critical functions to generate dynamic call graphs to uncover different runtime scenarios, taking into account the two layers and two psychoacoustic models used by the encoder, and associated specialization code. In a second stage, we use LARA aspects to instrument the code to monitor the application using API calls to hardware timers. Figure 2 illustrates a LARA aspect responsible for inserting API calls to hardware timers to measure the number of clock cycles of each call in a function. In our example, we wish to instrument only functions that contribute more than 10% of the overall execution time of the application. To determine the share of execution time of each function, this aspect requires the application to be first profiled (e.g., using GNU *gprof*) and the profiling results to be subsequently translated to a LARA format so that they can be used by the timing aspect presented in Figure 2. The instruction call profiling_module is responsible for loading the profiling results. Once the profiling results are loaded (via the import statement), they become available

```
import gprof_module;
aspectdef TimingStrategy
  call profiling_module; // call aspect that represents profiling information
  BeginOfFile: select file.($file_fstmt=first) end //select first statement in file
  CallsToFunctions: select file.function.call end //select function calls
  BeginOfFunc: select file.function.($func_fstmt=first) end //select first stmt in each fn

  apply to BeginOfFile::CallsToFunctions::BeginOfFunc // natural join operation
    $file_fstmt.insert before %{#include <time.h>}%;
    $func_fstmt.insert before %{time_t start,end;}%;
    $call.insert before %{printf("starting timer!\n"); time(&start);}%;
    $call.insert after %{time(&end);printf("elapsed time: %.2lfs\n",difftime(end,start));}%;
  end
  condition $call.time > 10 end
end
```

Figure 2. LARA aspect for timing function calls contributing with more than 10% of the execution time (using previously provided information).

DOI: 10.1002/spe

through the use of attributes associated to join points of type function and call. The condition instruction restricts the instrumentation to functions with a share of execution time larger than 10% ($call.time >10), which, as previously stated, is the result of profiling the code and translating the profiling results to a LARA report format. The instrumentation code, defined in the apply section, inserts the required header files, the declaration of the start and end time_t variables, the code to sample the time before and after the function call, and the code to report the measurements. These code elements are added at different source code points as specified by the LARA aspect.

In a subsequent stage, additional code is automatically inserted using another aspect to acquire the following: (i) runtime information about data exchanged between noncritical and critical functions and (ii) the number of iterations of the loops in these critical functions. When targeting an embedded system with a GPP coupled to hardware accelerators, one can explore the following two mapping scenarios: (i) mapping the entire code to the GPP leveraging software compiler optimizations or (ii) generating an application-specific architecture derived from partitioning the application between the GPP and reconfigurable hardware resources. Each of these mapping scenarios may require different strategies in terms of how the computation is partitioned between target components (e.g., GPP vs. reconfigurable hardware) and consequently how data are organized. Furthermore, once data and computations have been partitioned, data communication between the components and their subsequent synchronization need to be considered and included in the mapped code.

Once data partitioning and mapping are defined, the computations may still be subject to further transformations. For instance, they can be subject to a wide variety of transformations offered by software/hardware compilers [10], including 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 memories. With respect to computations mapped to reconfigurable hardware accelerators, there is a wider range of compilation and synthesis options that can be exercised that further increase the complexity of this mapping process. For example, local arrays can be mapped to the distributed memories available in the reconfigurable hardware device. Loop transformations can then be used to expose concurrent accesses to arrays. To achieve this mapping result, a developer may define a strategy to combine loop transformations such as unroll-and-jam followed by a specific mapping of array variables to internal storage resources [2].

In addition, although not discussed in this article, our compilation and synthesis approach also supports DSE strategies [11].

3. THE LARA ASPECT-ORIENTED PROGRAMMING APPROACH

In AOP, crosscutting concerns are expressed as aspects and are merged, or woven, with traditionally encapsulated functionality [6, 7]. The main concept of aspects can be described as: 'In program P, whenever condition C arises, perform action A' [12]. Associated with AOP are the notions of pointcut, join point, and advice [6, 7]. A pointcut designator exposes join points, which are points in the program execution. The areas in program text where join points may originate (statically or dynamically) are named as *join point shadows* [13]. An advice defines the actions to be performed for each join point exposed by the pointcut designator. Pointcuts can match join points dynamically at runtime or statically at compile time.

In AOP, a join point model defines the artifacts source of join points for a given programming language. A typical join point model includes function calls and read/write operations on variables. Our approach considers a join point model that captures most constructs (e.g., loops, conditionals, scalars, and accesses to arrays) found in C in order to support actions that target such code artifacts. LARA includes a join point model with both syntactic (structural) and semantic join points.

An important concept of AOP is the notion of weaving. With LARA, the weaving process combines, in an automated fashion, functional and nonfunctional concerns leading to the desired implementation. There are several benefits to the weaving process as pursued by the LARA-guided design flows:

- It allows nonfunctional concerns to be developed and maintained independently from the original application source code. This decoupling promotes a clean separation between the

DOI: 10.1002/spe

algorithmic specification and nonfunctional descriptions leading to a cleaner and thus easier to maintain source code base (Figure 3(a)).
- LARA aspects can specify strategies that capture a set of transformation steps to achieve different NFRs thus leading to potentially distinct implementations. Aspects can be applied and updated on the basis of different types of requirements without directly affecting the original source code. This feature substantially improves overall portability and application code maintainability (Figure 3(b));
- Aspects can be developed independently from application source code and therefore reused in the context of multiple applications. This reuse of aspects allows nonexpert developers to exploit specialized transformations and strategies geared toward specific target architectures, thus substantially promoting productivity and portability across similar target architectures (Figure 3(c));
- The ability to specify generic and parameterizable aspects in LARA is particularly useful for describing hardware-based and software-based transformation patterns and templates, thus facilitating DSE (Figure 3(d)). Examples of aspect parameters include application-specific and domain-specific information.

The LARA approach supports a *generalized weaving* process in which *actions* are tied to the program code base, to its intermediate representations or to the target system architecture. Another key innovation of our approach lies in bringing together, in the same framework, various types of transformations and operational aspects in the mapping of computations to embedded systems. Specifically, LARA allows developers to specify, in a broad sense, different types of concerns. Those concerns are mainly addressed by the following concepts:

- Monitoring: specification of monitoring features, such as current value of a variable or the number of items written to a specific data structure, providing insight for the refinement of other aspects.

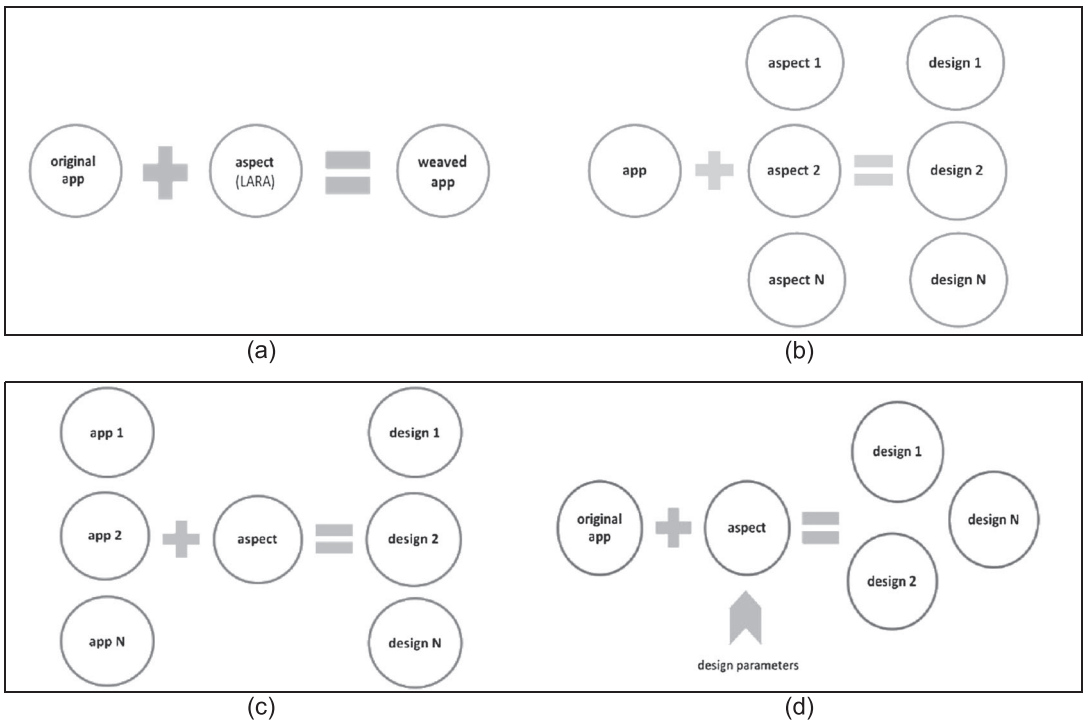


Figure 3. The LARA aspect-oriented programming approach: (a) weaving process decoupling function and nonfunctional specifications; (b) multitarget design; (c) aspect reuse; and (d) aspect parameterization for design space exploration.

DOI: 10.1002/spe

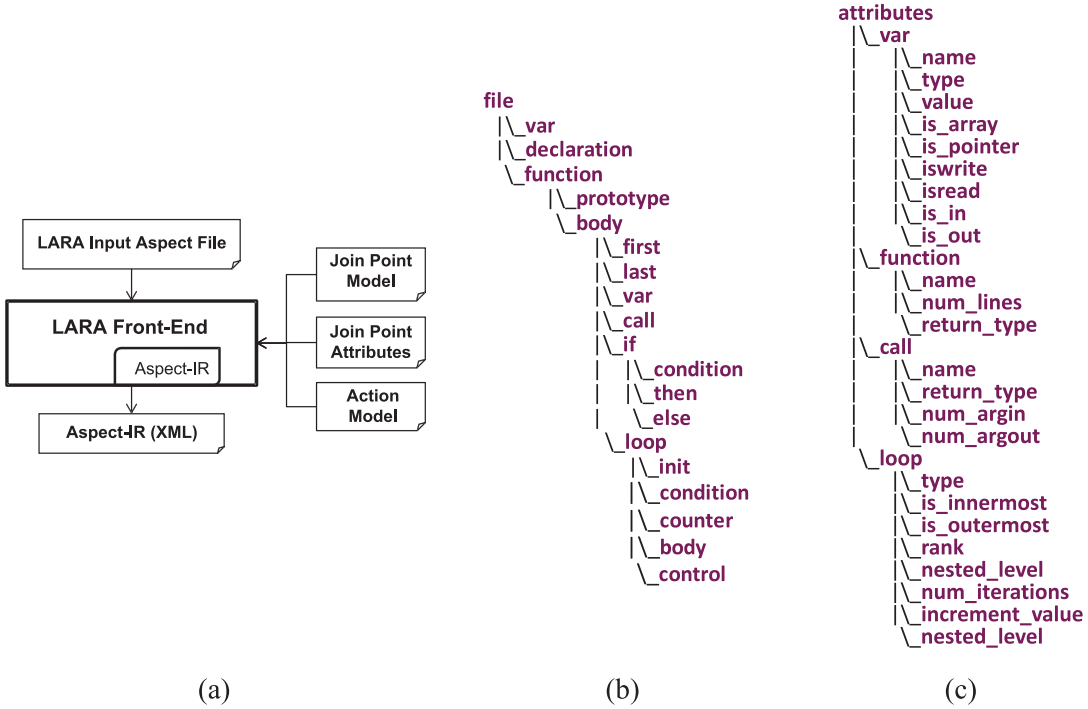


Figure 4. (a) LARA front-end and its input models: (b) excerpt of the join point model definition for C programs and (c) excerpt of the attribute model.

- 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 mapping and compiler optimization strategies, which embody mapping actions to guide tools to perform specific implementation decisions (e.g., mapping arrays to specific memories and applying loop unrolling to loops meeting certain conditions).

LARA aspects are processed by a LARA front-end component (depicted in Figure 4(a)), which converts a LARA aspect file into an executable specification in XML called the Aspect-IR. The front-end requires three specification files: (i) the join point model representing the points of interest in the input programming language; (ii) the join point attributes defining properties associated with each join point type[‡]; and (iii) the action model describing each possible action that an aspect can perform on a join point. We describe each of these models next.

3.1. Join point model

The program elements exposed to the weaving process are specified in the join point model. The join point model specification describes join point types and their hierarchy and can be used to validate pointcut expressions (also known as *pointcut descriptors* and as *pointcut designators*). Rather than hardcoding types in the LARA grammar, this approach allows the model to be easily updated and expanded to other target programming languages, tools, and platforms. 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 (e.g., Ref. [15]) and with different system components and architectures. Figure 4(b) and Table II show an excerpt of the join point model currently used for C programs. As an example, the join point *loop* has as its predecessor the *body* of a *function*, followed by the *file* it belongs to, and as its successor *init*, *condition*, *counter*, and *body* elements.

[‡]Some of the join point attributes supported can be seen as metaproperty attributes, and aspects using those attributes can be seen as metaproperty aspects in a similar way of the concepts presented in Ref. [14].

DOI: 10.1002/spe

Table II. Some join points and some of the possible attributes.

Join point	Attribute	Description	Examples
sym	name	Name of the variable	'x'
	type	Type of the variable	'int []'
	scope	Whether it is global, parameter or local	'local'
	num_reads	Number of times the symbol is accessed	10
	num_writes	Number of times the symbol is defined	15
	decl	Returns the join point statement that declared the symbol	<join point decl.>
	is_pointer/is_array	Whether the symbol is declared as a pointer or an array	False/true
	array_dim	Number of array dimensions (0 if it is scalar)	2
	shape	Returns a string with the size of each dimension if known	'[2] [14]'
var	name	Name of the variable	'x'
	type	Type of the variable	'int'
	is_array/is_pointer	Whether the variable is declared as an array or a pointer	False/true
	is_in/is_out	Whether the variable is being used or defined	True/false
	value	Value of the variable (if known at compile time)	10
	iswrite/isread	Whether is a write or a read to a var	False/true
	sym	Returns the join point symbol associated to the variable	<join point sym>
loop	type	Whether the loop is for, while, or do-while	'for'
	num_iterations	Number of loop iterations (−1 if unknown)	15
	is_innermost	Whether the loop is innermost in a loop nest	False/true
	is_outermost	Whether the loop is outermost in a loop nest	False/true
	increment_value	Increment step in each for-loop iteration	1
	nested_level	Returns the level in a loop nest	2
	rank	Returns the nested levels of the loop and its ancestors	'1 : 2'
call	name	Name of the function invoked	'sqrt'
	is_pointer	Whether it is invoked through a function pointer	False/true
	num_args	Number of arguments	3
	type	Function return type	int

The join point model considered by the LARA language includes both syntactic (structural) and semantic (behavioral) join points, that is, pointcuts that can consider not only program structures but also behavior. Syntactic join points refer to points of the code (or of the execution flow of the program) where the selection is only based on the program syntax, for example, loops, nested loops, and functions. Semantic join points are related to elements of the program where the selection is based on their meaning. A scalar variable of type *int*, an array variable, are examples of semantic join points. Possible join points related to behavior only known at runtime need to be explicitly translated to code responsible to verify the behavior at runtime. For example, a selection of a variable when it has the value 4 might be only resolved at runtime and specific verification code needs to be included in the aspect (i.e., the current LARA weaver does not generate the code needed to verify at runtime this behavior). LARA is, however, sufficiently modular and powerful to allow developers to specify native aspect code to verify complex behavior of the application on the selected join points.

The join point attributes, described in the following section, are the artifacts that can be used to define pointcut expressions considering static and/or semantic join points.

3.2. Join point attributes

The attribute model used by LARA contributes to a powerful selection and expressive mechanism as attributes can supply LARA aspects with system properties, and results from profiling, compiler analysis, or estimation tools, among others. Attributes expose information of each join point, which can be obtained by the weavers and/or can influence the use of actions as part of a strategy. Figure 4(c) illustrates examples of attributes associated with join points supported by our current join point model. For example, the join point type *var* has properties including its *name*, *data type*, and *kind* (e.g., array and scalar). Another example is the information we can obtain about a loop. For example, with the *number of loop iterations*, an aspect can decide whether to perform loop

DOI: 10.1002/spe

unrolling. On the other hand, with *is_innermost*, *is_outermost*, *nested level*, and *rank* attributes, an aspect can identify and select specific loops.

In the current implementation of LARA weavers, attribute values are acquired at compile time or using previously generated reports. 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 aspect action and also as arguments for the apply sections of an aspect. Join point attributes also provide a mechanism for exchanging information between tools in a design flow. For instance, the report data output from *gprof* is translated to LARA aspects by defining the values of join point attributes.

As with the join point model, the attributes specification file allows the aspect language to be updated and expanded more easily.

3.3. Action model

The action model specifies all actions that can be applied to join points. The most representative types of actions are the following:

- Insert: allows arbitrary code to be inserted before, around, or after a specific join point. This action is mainly used for instrumenting, monitoring, and adding functionalities if needed.
- Define: allows attributes to be modified.
- Optimize: allows a number of compiler transformations and optimizations (e.g., loop unrolling, function inlining/outlining, scalar replacement, and loop fusion) to be performed on a specific set of join points.

The action model is specific to a design flow and makes the LARA front-end aware of the actions, tools, and arguments possible to be used. The action model for each compiler optimization defines the name of the compiler engine and the possible parameters such as *loopunroll* (with unroll factor) or *tiling* (with size of the block). 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.

3.4. LARA examples

In this section, we present two practical examples of LARA aspects: counting loop iterations and timing applications.

3.4.1. Counting loop iterations. Determining the number of iterations of a loop at runtime is a key metric for the applicability of specific loop transformations. Figure 5(a) illustrates an aspect, which

```
aspectdef lcount
  select file.first end
  apply $first.insert before %{ #include "profiler_utils.h" }%; end //add profile header
  select function.loop.body.entry end
  apply $entry.insert after %{ profile_count("loop", "[[$loop.uid]]"); }%; end
  condition $loop.is_innermost end
  select function{name=="main"}.exit end
  apply insert before %{ profile_report("loop"); }%; end // report monitor
end
```

(a)

```
… int f1() {
  for (int i = 0; i < N; i++) {
    profile_count("loop", “uid124512”);
    for (int j = 0; j < M; j++) {
      profile_count("loop", “uid65123”);
    … } … }…
} …
```

(b)

Figure 5. Adding loop count monitoring for every innermost loop in the application: (a) LARA aspect and (b) excerpt of a woven code (inserted code is highlighted).

DOI: 10.1002/spe

instruments every innermost loop using two pointcut expressions. The first pointcut expression is used to insert the required *#include* statement, and the second pointcut selects the entry for each innermost loop, in which the statement to dynamically acquire the number of iterations is inserted. The loop *uid* attribute returns a unique identifier for every join point and in this case is used to identify each selected loop and to assign a unique location that stores the number of iterations (C code not shown and present in the function *profile_ count*). An example of the code after weaving is shown in Figure 5(b).

3.4.2. Timing applications. Developers often need to time their applications, which usually requires instrumenting the original application source code. One can describe this concern using LARA in a way that the aspect can be reused with other applications, while also reducing code pollution in the original source. Figure 6(a) illustrates an aspect that instruments the main function and adds the necessary code to time the functions of the application including loops (Figure 6(b)). It contains four pointcut expressions. The first selects the first statement of every file to insert the required include statement. The second, *Entry*, and the third, *Exit,* select the first and the return statements found in the functions, respectively. We join these two pointcut expressions to access these join points in one apply section and insert the required code to start and end the timer measurements for each function. The last pointcut selects the last statement executed before the return from the main function to insert invocations to the *profile_timer_report* function responsible for reporting the results.

As the LARA approach allows users to specify their own instrumentation strategy, users can deal with the possible instrumentation overhead by specifying strategies that instrument only specific parts of the application source code. For instance, when instrumenting loop nests, each loop in a nest can be instrumented and thus timed separately while still allowing for aggregated measurements a posteriori. Furthermore, one can measure the instrumentation overhead and subtract it from the timing measurements. These instrumentation schemes can be easily described in LARA, and their effectiveness depend on the profiling support given by the tools, operating system, and by the target architecture.

```
aspectdef Timer
  select file.first end  // select the first (global) statement of the file
  apply $first.insert before %{ #include "profiler_utils.h"}%; end //add profile header
  Entry: select function.entry end // select entry of all functions
  Exit: select function.exit end // select exit of all functions
  apply to Entry::Exit
    $entry.insert after %{ profile_timer_begin("[[$function.name]]"); }%;
    $exit.insert before %{ profile_timer_end("[[$function.name]]"); }%;
  end
  condition $function.has_loops end
  select function{“main”}.exit end // select last statement of the main function
  apply $exit.insert before %{ profile_timer_report(); }%; end
end
```

(a)

```
#include "profiler_utils.h"
…
int f1() {
  profile_timer_begin("f1");
  … for(…) {…} …
  profile_timer_end("f1");
} …
int main() {
  …f1();…
  profile_timer_report();
  return 0;
}
```

(b)

Figure 6. Timing functions of the applications with loops: (a) aspect that instruments the code and (b) code of the application after weaving (inserted code is highlighted).

DOI: 10.1002/spe

4. LARA LANGUAGE DESCRIPTION

In this section, we briefly present the LARA language (described in detail in Ref. [16]). LARA borrows syntactic and semantic elements from the JavaScript programming language [17]. In particular, many of the programming constructs used in LARA comply with the standardized ECMAScript scripting language (ECMA-262 specification and ISO/IEC 16262) [18]. Examples include arrays; loop constructs, that is, *for*, *while* and *do..while*; and conditional constructs, that is, *if* and *switch*.

In order to address concerns (e.g., NFRs), developers can specify strategies defined by one or more LARA aspects. Such aspects describe a sequence of weaving actions using an elaborated control flow execution. In order to support strategy composition, we include a mechanism that allows aspects to be invoked within other aspects. This mechanism is implemented by the *call* instruction (see Figure 2 for an example). The following sections describe the main structure of a LARA file and the principal components of the LARA aspects (Figure 7(a)).

4.1. Aspect definition

A LARA aspect file is composed by three main sections: import declarations, 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, contains the specification of each aspect. Finally, the code definition section, also optional, includes code, possibly with parameterized features, to be injected into the application source code.

An aspect definition (as depicted in Figure 7) is declared using the *aspectdef* keyword. Here, developers can define pointcut expressions and also the actions to take place over the selected join points. In LARA, an aspect definition consists of three dependent *weaving* sections: *select*, *apply*, and *condition*. For instance, the *apply* section can be associated with one or more pointcut expressions (*selects*). A *condition* instruction can be used to enable/disable *apply* executions, that is, actions over a join point. In general, there can exist various *apply* sections to the same select and *apply* sections associated with more than one *select* section.

Each aspect definition begins with the declaration of input and output parameters. These parameters are used to pass values to aspects and to return values from aspects, respectively. Each aspect has four additional optional sections for: declaring variables, declaring functions, code to initiate the

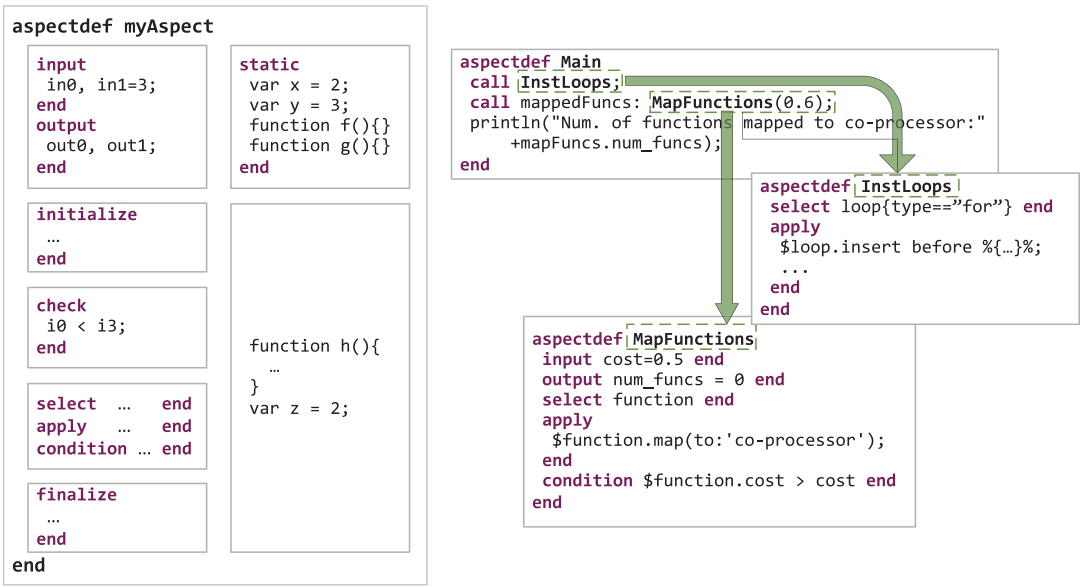


Figure 7. (a) The sections of a LARA aspect and (b) an example of aspect invocations.

DOI: 10.1002/spe

execution (*initialize*), and code to terminate the execution (*finalize*) of the aspect. Each aspect has a number of default attributes that store information about the execution of an aspect and include: *numapplies* (number of times the apply sections have been executed) and *changed* (if there were changes to the input program produced by actions, possibly only reflected in its intermediate representation).

Figure 8 depicts an aspect that inserts a *printf* statement immediately before each function call. To capture the intended join points a *select* statement is used. The *select* statement can be identified by a label (in our example, allFunctionCalls) and includes a pointcut expression that defines the join points to be captured. The pointcut expression represents a path (join point chain) in the join point model (Figure 4(b)), and it is validated by the front-end using the join point model specification. Pointcut expressions do not need to be specified using the complete paths as reflected in the join point hierarchy, as the front-end computes the entire path using the join point model specification as a reference. Also, the pointcut expression can use join point attributes to filter the selection.

One type of action associated with a pointcut expression is code insertion (see *insert* statement at line 4 in Figure 8) with the option to use values of join point attributes. As shown in the example in Figure 8, the code section between tags *%{* and *}%* in line 4 allows the use of parameters (between *[[* and *]]*) that are replaced by the weaver with the corresponding values. Code insertions can be carried out *before*, *after*, or *around* the join point. Target code can be placed between tags *%{* and *}%* after the *insert* command, as depicted in Figure 8, or in a separated LARA section called *codedef*. The LARA front-end does not take actions over the code by itself, and therefore, the insertion action is passed to the weaver.

Other supported actions include compiler 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* action inserts the body of the function to inline on the location of the function call.

In addition to setting the values of join point attributes without directly affecting the representation of a program, our current action model also includes the *define* action (*def* keyword). This action leads, whenever possible, to modifications of the program representation (e.g., defining the data type of a variable).

To control the scope of *apply* statements, developers can use conditions. Conditions can be regarded as logical expressions that support pointcut expressions to refine the join points they address and define the triggering conditions for actions specified in an *apply* statement. Join point attributes can also be used to build condition expressions. Examples of actions dependent on specific conditions are the following: (i) apply *loop unrolling* only if the loop is of type for and has at most eight iterations; (ii) insert code before a function call if it returns a float; and (iii) *inline* a function if it does not contain loops. LARA also allows the use of conditional statements (*if*, *if-else*, and *switch* statements) in the code of *apply* sections, and they can be also used to limit the actions over selected join points.

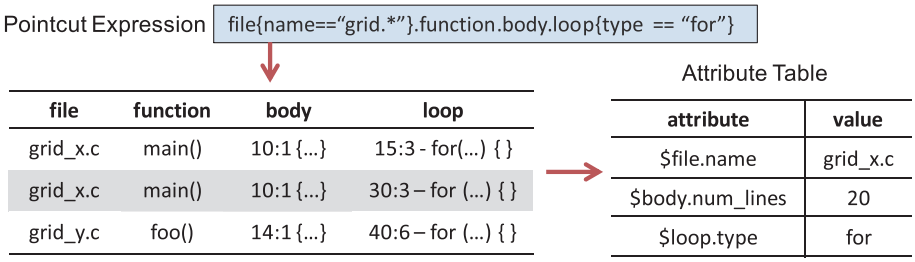
4.2. Join point chains

In LARA, pointcut expressions reflect the hierarchy defined in the join point model (Figure 4(b)). For instance, in Figure 9, the pointcut expression has four elements connected in a chain: the *file* captures all files with names starting with *grid*, the *function* includes all functions enclosed in

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

Figure 8. Example of a simple LARA aspect code that performs instrumentation.

DOI: 10.1002/spe



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 9. 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.

those files, the *body* reflects the code of those functions, and the *loop* captures for-type loops in those functions. This hierarchical mechanism has similarities to the *within/withincode* used in AspectJ [19, 20], but in this case, we are able to address other program elements besides fields and methods of classes.

To access the join points resulting from the evaluation of a pointcut expression, we use the *$* operator with the pointcut element identifier. In the example in Figure 9, *$file*, *$function*, *$body*, and *$loop* refer to chained join points. We can use these join points and their attributes as part of an action or a condition. Join point identifiers enable access to join point attributes, for example, given the pointcut expression *file.function.loop*, *$file.name* refers to the name of the file where a particular *$loop* join point can be found.

An action can also target any element associated with a join point chain. By default, actions are performed on the last join point in the chain specified by the pointcut expression. However, LARA allows the specification of actions (e.g., *insert*) using the identification of other join points in the chain. In the following example, the first action inserts code after the *$loop* join point (the last join point in the chain), whereas the second action inserts code before the *$body* join point.

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

4.3. Composite pointcut expressions

Multiple pointcut expressions can be composed using logical and other special operations. For instance, they 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 to their common point element identifiers. In the example of Figure 10, 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 functions. The use of **::** allows to apply actions and access attributes on join points that are found in different hierarchical branches (*$loop* and *$first* in this example) of the join point model. As another example, Figure 2 presents an aspect using the natural join operator applied to three pointcut expressions.

```
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;

Figure 10. Example of a pointcut expression using the natural join operator '**::**'.

DOI: 10.1002/spe

4.4. Control flow pointcuts

Pointcuts describing control flow can be important to apply actions when the program execution reaches specific execution points such as ones exposing sequences of function calls. Control flow information is usually required for various concerns, such as debugging, testing, maintenance, and program comprehension. LARA does not provide native support for pointcut expressions/designators specifying interprocedural control flow. Instead, control flow is supported in LARA through the explicit use of code instrumentation. For example, a LARA implementation of the AspectJ *cflow* and *cflowbelow* [19, 20] includes the explicit inclusion of code necessary to maintain a stack on function calls. This can be transparent to the developer using the modularity provided by LARA. For instance, the pointcut designator *(and (pcalls f) (pwithin g) (cflow (pcalls h)))* given at Ref. [21] 'indicates that the piece of advice to which this pointcut designator is attached is to be executed at every call to procedure *f* from within the text of procedure *g*, but only when that call occurs dynamically within a call to procedure *h*'. Below is a possible LARA implementation that calls an aspect that instruments the code with the required call stack functionality, including the use of *stackContains("h")*, a function provided by the *InsertCallStack* aspect.

```
aspectdef CFlowExample
  call InsertCallStack();// Instrument the code with the call stack.
  select function{name == "g"}.call{name == "f"} end // select the "f" calls in "g"
  apply $call.insert before %{ if(stackContains("h")) { /* code to insert */ }}%; end
end
```

4.5. Exposing program variables

As part of LARA's join point model geared towards the C language, program variables can be accessed in two ways: from the program scope (commonly associated to a symbol table) and as part of the source code. Most programming languages provide the notion of scope that defines the lifetime of a programming variable.

To access variables that are visible from a particular code block, we use the pointcut element *sym*. For instance, to select all symbols available to the function, we use the following *select* statement: *select function.body.sym end*. In this example, the join points associated to *sym* expose all symbols that are visible to the top-level function scope, including those available in the global scope and function parameters. Rather than referencing specific locations of the source code, these join points expose elements from symbol tables. In contrast, variables can be a part of an expression inside code statements. We can select these variables using the *var* pointcut element. For instance, the following *select* statement finds all variables 'x' found in a function: *select function.var{name=="x"} end*. In this case, multiple *var* join points corresponding to the same variable can be exposed as a result of a *select* statement to select different parts of the code. Table II includes some of the attributes supported by the *sym* and *var* join points.

As an example, consider the following aspect that monitors the ranges of single-precision floating-point variables in the application. This aspect finds all writes to floating-point scalar variables. For each of these writes, the aspect instruments the code with a monitoring function, *prof_range_float*, which considers the new value written. Hence, during execution the range values (min and max) of each variable are acquired. This simple strategy can be useful, for example, for further hardware design optimizations by customizing the word lengths of variables and expressions.

```
select function.var end
apply
  $var.parent_stmt.insert after %{
    prof_range_float("[[$function.name]]:[[$var.name]]",  (float) [[$var.name]]); }%;
end
condition $var.is_scalar && $var.iswrite && $var.type=="float" end
```


DOI: 10.1002/spe

4.6. Exposing program loops

In LARA's current join point model, all types of loops are captured by the *loop* pointcut element. This includes for, while, and do-while loops (identified by the type attribute). Loop join points support a number of specific attributes (Figure 4(b) and (c) and Table II) such as the *nested_level* and the *rank*, which can be used to select loops in loop nests in different ways as shown by the pointcut expression examples presented in Table III. In general, chained join points include not only the link between parent and immediate children but also all combinations between parent and descendants in the nested program structure. Using attributes such as *nested_level*, one can filter unwanted combinations.

Loop join points can also link to their program structures that depend on the type of loop. In the pointcut chains, loops can link to body, condition, initiation, and counter. These join points are useful to access loop structures.

4.7. Exposing program conditionals

Branches can be selected using the *if* pointcut element that locates *if* statements in the application source code. In a *select* statement, the *if* pointcut element can be chained with the following pointcut elements: *cond* returns an expression corresponding to the *if* condition, *then* returns the block of code corresponding to the *true* branch, *else* returns the block of code corresponding to the *false* branch, and *body* returns both *true* and *false* branches.

The following LARA aspect shows how one can monitor branch frequencies. The aspect instruments all if statements by inserting a function call (*profile_branchfreq*) at the beginning of each branch (true and false) to monitor the number of times a branch is executed. The arguments of this monitoring function are as follows: (i) an identifier of the branch; (ii) an identifier of the (parent) conditional statement; and (iii) a label that describes the branch. To identify the branch and the conditional statements, we use the corresponding *uid* attribute that returns a unique number identifying the program element. The label identifying the branch is obtained with the *branch_cond* attribute that returns the C condition associated with that branch. Branch frequency monitoring may help

Table III. Examples of pointcut expressions for loops.

Code example	Pointcut expression	Chain selection	Description
A. for(i=0; i < X; i++) B. for(j=0; j < Y; j++) C. for(k=0; k < Z; k++) D. for(l=0; l < W; l++)	Select loop end	A, B, C ,D	Select all loops
	Select ($l1 = loop). ($l2 = loop) end	A → B, A → C, A → D B → C, B → D, C → D	All combinations of two loops in a loop nest. Note: A → B corresponds to $l1 = A and $l2 = B
	Select ($l1 = loop). ($l2 = loop).($l3 = loop) end	A → B → C, A → B → D, A → C → D, B → C → D	All combinations of three loops in a loop nest
	Select loop {nested_level==1} end	B	Only one loop in the loop nest satisfies this condition
	Select ($l1 = loop) {nested_level! = 0}. ($l2 = loop) end	B → C, B → D, C → D	All combinations of two loops that does not include the outermost loop
A. for(i=0; i < X; i++) B. for(k=0; k < Z; k++) C. for(k=0; k < Z; k++) ... D. for(j=0; j < Y; j++)	Select loop end	A, B, C, D	Select all loops
	Select loop{rank == 2} end	D	Second top-level loop
	Select loop {rank == 1.1} end	B	First loop in the second level of the first nested loop structure
	Select loop{rank == 1.2} end	C	Second loop in the second level of the first nested loop structure

DOI: 10.1002/spe

developers and tools to determine the branches more biased and that could benefit, for example, from more hardware resources and optimizations.

```
aspectdef branch_frequency_monitor
  select function.if.body.begin end
  apply
    insert after %{ profile_branchfreq_monitor("$body.uid]]",
                    "[[$if.uid]]", "[[$function.name]]:[[$body.branch_cond]]");}%;
  end
end
```

4.8. *Exposing program function calls*

Function invocations or calls can be selected using the *call* pointcut element. This pointcut element is usually chained with the *args* pointcut element, which returns all argument expressions associated with the function invocation. Examples of attributes associated to *call* join points are listed in Table II.

The following aspect segment reports the execution time of all non-system function calls that are not invoked via pointers. The strategy includes instrumenting each candidate function call before and after to compute the elapsed time. Such monitoring strategy can be used to find critical sections in the program and, unlike conventional profiling tools, allows fine-grained control over what regions of code to profile.

```
select function.call end
apply
  $call.parent_stmt.insert before %{
    static XTime  timeStart_[[$call.uid]], timeEnd_[[$call.uid]];
    XTime_GetTime(&timeStart_[[$call.uid]]); }%;
  $call.parent_stmt.insert after %{ XTime_GetTime(&timeEnd_[[$call.uid]]);
    printf("time elapsed for  '[[$call.name]]': %llu ns\n", (timeEnd_[[$call.uid]] -
            timeStart_[[$call.uid]]) * 10); }%;
end
condition !$call.is_sys && !$call.is_pointer end
```

4.9. *Select–apply semantic*

The LARA weaving process operates with join point shadows (i.e., program elements in the source code) that are selected statically according to pointcut expressions. The code in *apply* and *condition* sections is statically executed. The exception is the code specified in an *insert* action that is injected in the application source code and is only executed during the execution of the compiled woven code. The decision to have an offline weaving process is because of the nonacceptable overhead that would result of an online weaving process and also because most weaving actions currently supported by LARA, for example, related to code transformations and code optimizations, are undesirable and/or impossible to perform at runtime.

There are cases however where specific actions are dependent of join point attributes with values only known at runtime. Figure 11(a) presents an aspect with an action that is dependent on the evaluation of a condition using two attributes (*iswrite* and *value*) of a join point related to variables used in a function. The attribute *iswrite* is resolved offline as it requires the inspection of code assignments to variables. However, the value of the attribute *value* is only known at runtime, unless the variables have statically known constant values. This condition may require the insertion of code responsible to evaluate the value of the variable during runtime, unless the specific value is obtained by a previous execution of the application and passed to the aspect (as in some of the cases presented in the industrial case study in Section 6). The aspect in Figure 11(b) uses an explicit insertion of that code and uses a condition statement fully resolved offline. When considering a static weaving process, as in our current case, and the evaluation of a runtime dependent value, the second aspect (Figure 11(b)) must be used.

4.10. *Invoking aspects and execution semantics*

The front-end requires a single LARA source file as input. The first aspect definition in this file is the one to be executed by default. All subsequent executions of aspects are performed explicitly by aspect calls. A call can invoke any aspect definition in that source file or in external files indicated

DOI: 10.1002/spe

(a)

```
aspectdef print1
  select function.var end
  apply
   insert after %{printf("Warn: %s exceeds 10!\n",[[$.var.name]]);}%;
  end
  condition $var.iswrite && ($var.value > 10) end
end
```

(b)

```
aspectdef print1
  select function.var end
  apply
   insert after %{
     if([[$.var.name]] > 10) printf("Warn: %s exceeds 10!\n",[[$.var.name]]);
    }%;
  end
  condition $var.iswrite end
end
```

Figure 11. Dynamic and static attributes: (a) an action depending on the value of a static and a dynamic attribute and (b) an action depending on a static attribute.

by the import statement. LARA supports two mechanisms for invoking an aspect. The first uses the traditional in-line 'call/return' procedure (no return value) semantics. In this case, we can use, assuming `myAspect` is an aspect module, `call myAspect;` or `call myAspect(1, num)` when passing arguments to the aspect. The second mechanism uses an object-oriented notation. In this mechanism, the program can create a named aspect object when instantiating the object and then explicitly invoke it with the call statement:

```
var A = new myAspect; // declaration of a myAspect instance identified by variable A
call A;  // call to a myAspect instance identified by variable A
…
call B: myAspect; // call to a myAspect instance identified by variable B
```

There is also the possibility to pass arguments in the declaration of the aspect instances (e.g., `var A = new myAspect(1, num)`) or when calling each specific instance (e.g., `call A(1, num)`). The input parameters of an aspect can also be accessed using the name of the aspect or of an instance of the aspect followed by '.', and the name of the input parameter such as: `A.factor = 1` or `A.numLines = num`. The values of output parameters can be accessed with the same syntax as the input parameters. For instance, in the following example, the *access* output parameter is used: `if(A.success) print ("aspect was successfully applied!\n")`.

Aspect invocation behaves as a function call in common imperative languages, with limited inter-invocation state sharing. Within an aspect definition, input and output parameters are treated as variables, which can be accessed and modified without restrictions. Global variables defined within the context of an aspect definition can be accessed and modified (retaining state) across all apply actions in that aspect but not across other aspects. Local variables defined within an *apply* section retain their values during the sequence of executions of the *apply* and are only visible in that section. As such, there are no side effects as a result of an aspect call.

All aspect input arguments are evaluated eagerly and passed by value. Aspects can only share state through aspect parameters and static variables.

The order of execution between select–apply statements inside a LARA aspect definition is dictated by a particular execution semantic. Currently, the order of execution of *select–apply* sections follows the textual order present in the input LARA aspect module.[§]

5. EXPERIMENTS WITH LARA USING CODE INSTRUMENTATION

In this section, we evaluate the effectiveness of LARA in capturing nonfunctional concerns related to application monitoring and apply these concerns automatically and systematically to a number of

[§]Although possible, there is no support for concurrency between one or more *select–apply* sections.

DOI: 10.1002/spe

benchmarks (Section 5.3). The nonfunctional monitoring concerns considered in this section, such as counting loop iterations, monitoring range values, and timing applications, correspond to tasks often carried out by developers by manually instrumenting their applications' code to extract runtime behavior, in order to understand which parts of code to optimize, for instance, by offloading to hardware accelerators. In this context, there is considerable effort in modifying existing applications to implement these concerns, affecting both productivity and maintainability.

This section aims to answer the following questions:

- Is LARA able to capture different monitoring concerns?
- Can a single monitoring concern described in LARA be reused in different applications?
- Can multiple monitoring concerns described in LARA be used in the same application code?
- Are we able to reduce the programming effort of monitoring concerns on existing applications when compared with manually implementing these functionalities?

5.1. Methodology

5.1.1. Design flow. The experiments described and reported in this section use the Harmonic [22] source-to-source compiler, which was enhanced to support LARA specifications [9]. As shown in Figure 12, the compiler accepts two types of source descriptions as inputs: (i) input application in C (C99 std. compliant) and (ii) LARA files capturing concerns in the form of aspects and strategies. For the experiments reported here, we rely on a LARA-based design flow structured as three major components:

- LARA front-end, which translates 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, which uses Harmonic to perform source-level transformations (C to C) including but not limited to code instrumentation and monitoring, hardware/software partitioning using cost estimation models, and 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, which includes the front-end, middle-end, and optimization phases, with the latter two common to both software and hardware partitions, which are target architecture

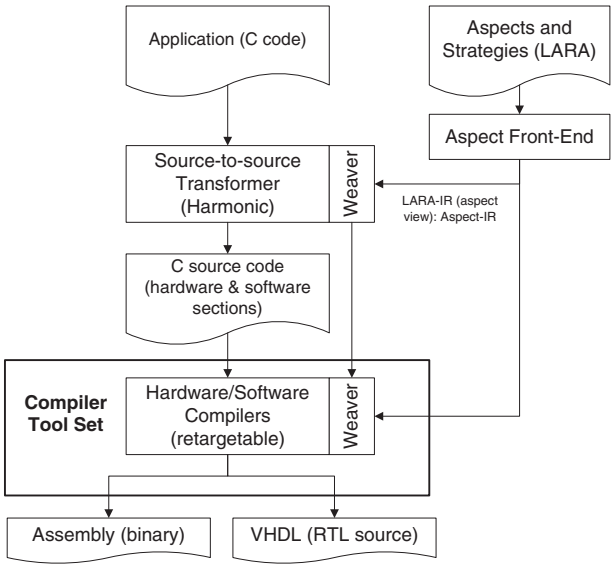


Figure 12. Typical LARA-based design flow.

DOI: 10.1002/spe

independent. This stage can be performed by any hardware/software C compiler or by a specific C compiler that is controlled by LARA aspects and strategies. The code generators of the compiler tool set may include assembly code generators for GPPs (software sections) and VHDL/Verilog generators for specific hardware cores.

A typical LARA-based design flow may also include additional 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.

5.1.2. Aspect metrics. To assess the impact of our approach, we use seven metrics: the number of join point shadows, lines of code (LOC), concern impact on LOC (CILOC), concern diffusion over LOC (CDLOC) [23], aspectual bloat [24], tangling ratio [24], and the function-level weaving coverage (percentage of functions that are affected by the given aspect). Table IV associates each research question to the metrics used. We explain each of these metrics next.

The LOC metrics referred herein are as follows: (i) the number of LOC in the original application; (ii) the number of lines of the woven application; and (iii) the number of lines in the LARA descriptions. These metrics provide a measure of complexity for both C and LARA descriptions. For instance, the aspects listed in Table VI have 23–34 LOC.

The number of join point shadows refer to the number of program locations where join points may originate.

The CILOC gives us the ratio between the original source code LOC and the woven code LOC on which concerns have been applied. This metric gives us a first intuition about the impact of using aspects. Thus, the smaller and further away the CILOC is from 1, the larger is the woven application in comparison to the input source code.

The CDLOC metric, on the other hand, measures the number of switch points (also called transition points) where code transitions from concern-specific (code introduced as a result of an aspect) to functional (original code) and vice versa [23]. Hence, this metric provides a measure of code intermingling. The number of transition points depends on the concern, as it may require the inclusion of only a block of statements per join point, or may require in more complex cases the inclusion of blocks of statements in different source code locations.

Another related aspect metric is the ratio between the CDLOC count and the woven code LOC, called the tangling ratio [24]. The tangling ratio gives us another measure of code intermingling, this one revealing the number of transition points per line of code in the resultant woven

Table IV. Aspect metrics using goal–question–metrics.

Goal	Question	Metric
To minimize the effort in applying nonfunctional concerns on existing applications	How much effort is required to analyze existing applications to apply a new concern?	LOC Number of join point shadows
	What is the impact of applying a concern in the application?	Number of join points CILOC Function-level weaving coverage
	What is the complexity of incoorporating a concern into the functional code?	CDLOC Tangling ratio
	Is it worth to manually implement a concern in a single application, or automatically through aspects?	Aspectual bloat
	Is it worth to manually implement a concern in a set of applications, or automatically through aspects?	Aspectual bloat reuse

LOC, lines of code; CILOC, concern impact lines of code; CDLOC, concern diffusion lines of code.

DOI: 10.1002/spe

Table V. Main characteristics of the benchmarks used.

Benchmark repository	Application/kernel	#files	#functions	#call statements	#loops	#innermost loops	#IF statements	#LOCs
REFLECT	3dpath (3DPP)	6	48	164	46	15	43	9,444
	stereo_nav (SN)	26	157	1,099	397	313	600	10,075
	mpeg_enc (MPEG)	15	154	1,152	261	165	443	9,022
	g729 (G729)	33	127	520	391	302	535	11,005
MiBench consumer	jpeg	54	391	1,745	531	372	1,471	24,866
	lame	32	230	5,718	509	355	1,182	16,031
	tiff2bw	34	375	6,958	515	377	1,532	22,887
	tiff2rgba	34	372	12,758	514	373	1,526	22,771
	tiffdither	34	371	7,797	510	372	1,540	22,735
	tiffmedian	34	377	6,990	558	395	1,589	23,516
MiBench network	dijkstra	1	6	26	7	5	10	163
	patricia	2	7	66	10	10	39	569
	CRC32	1	4	9	3	3	2	123
	sha	2	8	40	10	10	7	233
	blowfish	7	8	30	12	11	30	965

LOC, lines of code.

Table VI. Concerns and corresponding Aspects used in our experiments.

Concern (abbreviation used)	Aspect name	Goal	Aspect #LOC
Counting loop iterations	lcount	Print the number of iterations for each innermost loop in the application	23
Monitoring range values	vrange	Print the range values for each *int* variable	23
Monitoring memory allocation/ deallocation	hmem	Monitor heap memory allocation and deallocation (malloc, realloc, free)	28
Computing branch frequencies (bfreq)	bfreq	Generate a runtime branch frequency report	34
Computing a dynamic call graph (cgraph)	cgraph	Generate a dynamic call graph with estimated cost for VIRTEX5 and PPC	26
Timing applications (timer)	timer	Timing all functions that contain loops	28

LOC, lines of code.

application. 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 aspectual bloat [24] measures the efficiency of an aspect with respect to the woven code generated. This metric is computed by dividing the number of LOC 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.

In order to have a measure of both the efficiency of an aspect with respect to the generated woven code and its reuse, we include an extension to the aspect bloat that we named 'aspect bloat reuse'. The aspectual bloat reuse is calculated considering all the applications to which the same aspect has been applied and uses the following quantitative evaluation expression:

$$\text{aspectual bloat}_{(\text{with reuse})} = \frac{\sum \text{LOC}(\text{application after weaving}) - \sum \text{LOC}(\text{application})}{\text{LOC}(\text{aspects used})}$$

Large values of the aspectual bloat reuse mean high levels of reuse.

5.2. Application concerns

In this section, we present experimental results using the LARA language to capture nonfunctional concerns and a LARA-based design flow toolchain on three benchmark repositories. These

DOI: 10.1002/spe

repositories consist of two groups of applications from the MiBench benchmark suite [25] (consumer and network) and four real-life applications that have been provided by the two industrial partners in the REFLECT project. Table V presents the relevant characteristics of the benchmarks used. They consist of simple applications that are part of the MiBench Network group (from 1 to 7 files, 4 to 8 functions, and 163 to 965 LOC) and the more complex applications that are part of the REFLECT and MiBench Consumer benchmark sets (described in the next section).

5.2.1. Targeted benchmarks. The REFLECT benchmark set consists of two audio domain applications (an MPEG-2 audio encoder (MPEG) and a G729 voice encoder) and two avionics applications (SN and 3DPP). A set of concerns and application requirements were translated into LARA aspects. These concerns are associated 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 introduced in Table I a list of concerns for the MPEG application. 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 of the application to the target architecture. This, we believe, demonstrates the flexibility and wide scope of our approach.

As previously explained, the monitoring capability enables us to understand runtime behavior, such as the following: (i) the most executed paths in *if-then* constructs; (ii) nonvariant parameters in specific function calls that enable distinct specializations for the same function; and (iii) data ranges and accuracy that can be used to guide word length optimizations. The use of aspects related to (ii) allowed us to identify opportunities for function specializations in all the four applications. For instance, in 3DPP, we evaluated three specialized versions of the *griditerate* function, and in SN, we considered three possible specializations of a convolution function [9]. These specializations were important to achieve better performance as in most cases they were related to loop iteration counts, enabling full loop unrolling and strength reduction.

5.2.2. Aspects for application code analysis. We now consider a number of representative monitoring concerns (Table VI). They consist of measuring branch frequencies, monitoring range values (min and max) for variables in the program, monitoring function calls, monitoring the number of iterations of loops (see example in Figure 5), monitoring memory allocation and deallocation, and timing applications (see example in Figure 6). We expressed the concerns as LARA aspects (last column of Table VI presents the number of lines for the LARA aspect for each concern) and applied them to the four industrial applications previously introduced and to a number of applications of the MiBench consumer and network sets [25] (Section 5.3).

5.3. Evaluation

We apply the six concerns summarized in Table VI to the benchmarks briefly described in Table V. These concerns represent common monitoring aspects and include the following: (i) counting of iterations of selected loops (*lcount*); (ii) reporting of the range values of specific variables (*vrange*); (iii) monitoring of memory allocation and deallocation instructions (*hmem*); (iv) reporting of the frequencies of the execution paths taken considering *if* and *if-else* constructs (branch frequencies, *bfreq*); (v) generating a customized call graph with information related to execution costs (*cgraph*); and (vi) timing of all functions containing loops (*timer*).

Figure 13 presents the number of join point shadows selected when applying the LARA aspects listed in Table VI. A minimum of 5 (*CRC32* benchmark and *lcount* concern), a maximum of 16,170 (*jpeg* benchmark and *vrange* concern), and an average of 1,931 join point shadows were selected. These join points represent code elements in the source code in which we apply instrumentation. The large numbers of join points indicate a high manual effort if an automated approach was not used. This manual effort is actually larger as the actions related to the concerns are selective and need an analysis of each join point (including its context) in order to decide about the application

DOI: 10.1002/spe

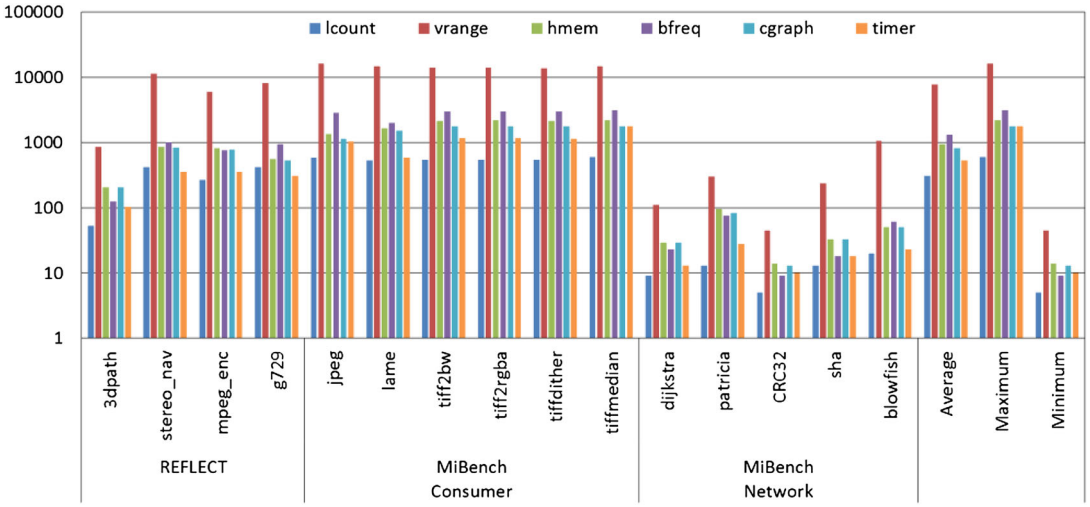


Figure 13. Selected join point (shadows) for each application considering each aspect.

of the actions. As expected, the *vrange* concern implies the highest number of join point shadows because of the high number of variables in the source code. The *lcount* concern has the lowest number of join point shadows as the number of loops in the source code is not as high when compared with other program elements.

The number of join point shadows is lower for the MiBench network benchmark set (minimum of 5, maximum of 1074, and an average of 84) as the programs included in this benchmark have from 163 to 965 LOC and have potentially a lower number of join points (Figure 13). The percentage of join point shadows selected for applying actions are however very similar. Figure 14 illustrates the averages related to those percentages when considering the three benchmark sets and the six concerns. A minimum of 0.2% (*stereo_nav* benchmark and *vrange* concern), a maximum of 100% (*patricia*, *CRC32*, and *sha* benchmarks and *lcount* concern), and an average of 49.9% of join point shadows were selected for applying insertion actions. In addition, Figure 14 shows the percentage of functions affected by the code insertion actions. The insertion coverage corresponds to a minimum of 0.6% (*stereonav* benchmark and *vrange* concern), a maximum of 88.3%

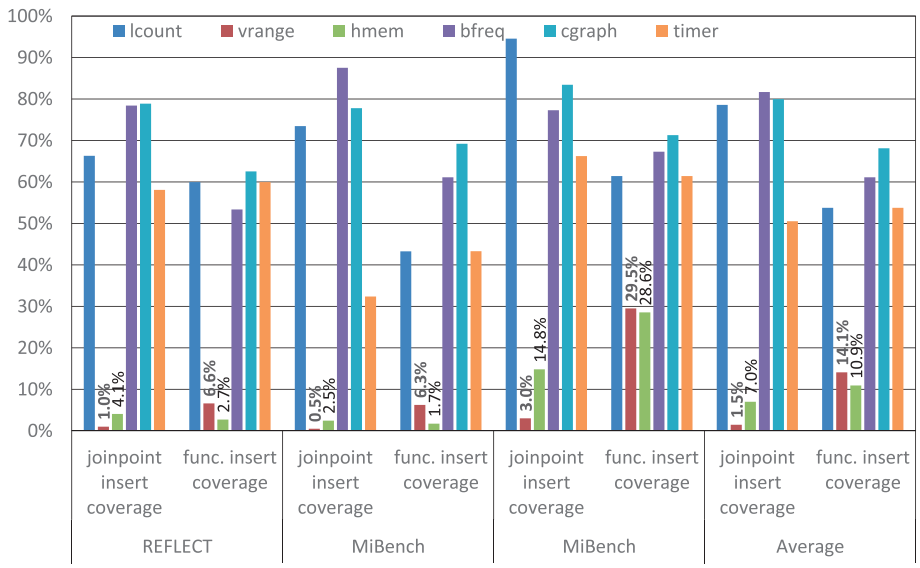


Figure 14. Averages related to the percentages of the selected join point (shadows) and of the functions where insert actions took place.

DOI: 10.1002/spe

(*dijkstra* benchmark and *bfreq* concern), and an average of 43.6% of the total number of functions in the three benchmark sets.

Despite a high number of join point shadows selected by the *vrange* and *hmem* aspects, the percentage of those join points to which code insertion actions have been performed is very low (1.5% and 7% on average for *vrange* and for *hmem*, respectively). With respect to *vrange*, this is a consequence of the fact that the monitoring concern only focuses on writes to scalar variables of type *int*, while the select statement captures all variable references, which are subsequently filtered by a condition statement. The percentage of functions where instrumentation takes place is 14% and 10% for *vrange* and for *hmem*, respectively.

Figure 15 shows the tangling ratios for the benchmarks considering the six concerns in Table VI. The highest tangling ratios occurred for *bfreq* and *cgraph* concerns as both the percentage of selected join points and the percentage of join points where actions were applied are high (around 80% on average) as can be seen in Figure 14. The lowest tangling ratios occurred for the *vrange* concern. For this concern, the selected number of considered join points is one of the highest (Figure 13), but the ones where actions are in effect applied is very low (1.5% on average) as can be seen in Figure 14.

The CILOC metric for the aforementioned benchmarks and concerns reflects a similar result as the tangling ratio metric. In particular, the inserted code is always smaller (a couple of code statements) when compared with the application code. The CILOC metric shows 4% (CILOC = 1.04) of code increase on average. The highest CILOC metric values of 1.15 and 1.14 were obtained for *dijsktra* and *patricia* considering the *cgraph* concern.

The CDLOC metric, on the other hand, reveals average values from 66 (for *hmem*) to 1242 (for *cgraph*), a minimum of 4 (for *vrange* and *hmem* when applied to *CRC32*) and a maximum of 5450 (for *bfreq* when applied to *tiffmedian*). The global CDLOC average is 762 that clearly indicates a high intermingling between application source code and code related to concerns.

Figure 16 shows the aspectual bloat values for our three benchmark sets and the six concerns from Table VI. We measured an aspectual bloat of 12.96 on average. Globally, the highest values of the aspectual bloat occurred for the *bfreq* concern when applied to the MiBench consumer benchmarks. For these benchmarks, the aspectual bloat values range from 51.56 to almost 79.47. In case of the MiBench network benchmarks, the highest values of the aspectual bloat occurred for the *cgraph* concern. For the REFLECT applications, the aspectual bloat values are higher for *cgraph* when applying this concern to *3dpath*, *stereo_nav*, and *mpeg_enc* and for *bfreq* in the case of the *g729* application. This is explained by the fact that *g729* includes a higher percentage of branch statements and a lower percentage of function calls, while *3dpath*, *stereo_nav*, and *mpeg_enc* have a lower average of statements per function that based on the number of functions imply a higher overhead of the code inserted per function call statement. The *hmem* concern has the lowest

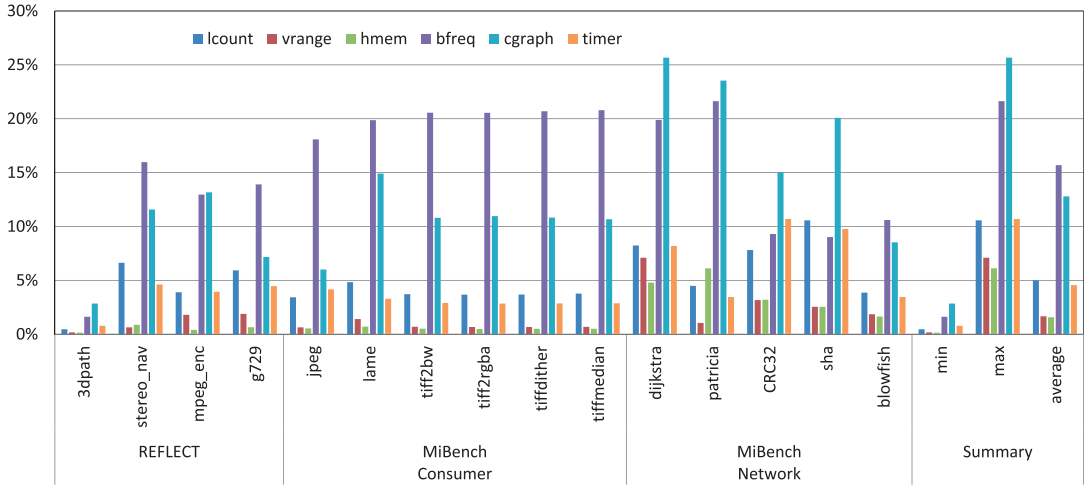


Figure 15. Tangling ratio for each application and considering each aspect.

DOI: 10.1002/spe

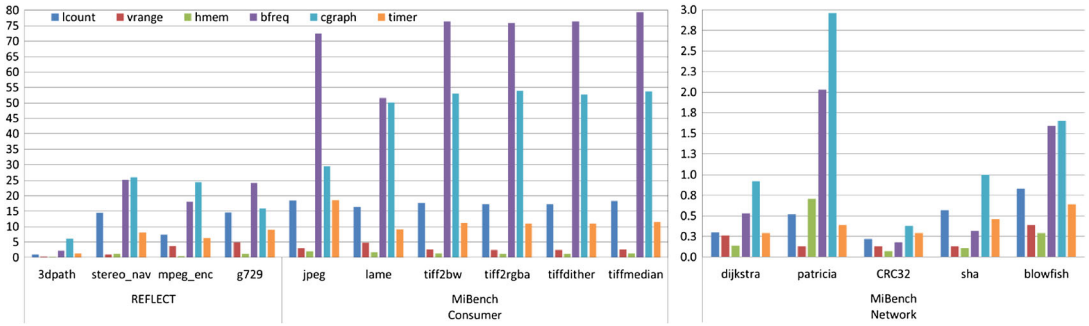


Figure 16. Aspectual bloat for each application when considering each of the six concerns.

aspectual bloat, followed by the *vrange* concern. This reduces the impact of the woven application source code as it has similar LOC values (0.22% on average for all the benchmarks used). The *vrange* concern also has a low impact on the woven code as the LOC metric increases only 0.38% on average. Both *hmem* and *vrange* have a low percentage of join points effectively used for weaving actions (Figure 14). In case of *hmem*, the aspect replaces invocations to standard memory management functions *malloc*, *free*, and *realloc* with wrapper functions that monitor memory allocation and deallocation operations. The CDLOC and tangling ratio values also point to the same conclusions.

Figure 17 shows the aspectual bloat for reuse. The measurements clearly show the high reusability of the aspects related to the concerns in Table VI. All these aspects can be reused with every C application and not just benchmarks used in this paper. The results show that the efficiency of the aspects in terms of woven code and reuse is higher for *bfreq* (506.79), followed by *cgraph* (372.4) and *lcount* (145.6). The *hmem* and *vrange* are concerns with lower efficiency. The MiBench network benchmarks exhibit lower efficiency (aspectual bloat values from 1.0 to 6.9) as was expected given the number of selected join points (Figure 13) and the percentage of those join points effectively used for instrumentation actions (Figure 14).

For the aspects used in these experiments, the execution time of the weaving process strongly varies with the size of the application but less with the aspect. The execution time varies from seconds (Mibench consumer) to minutes (around 12 min for MiBench networks).

5.4. Summary

With respect to the questions listed at the beginning of this section, the experiments provide strong evidence that: (i) LARA is able to capture monitoring concerns typically part of a development cycle targeting embedded platforms; (ii) a single monitoring concern described in LARA can be

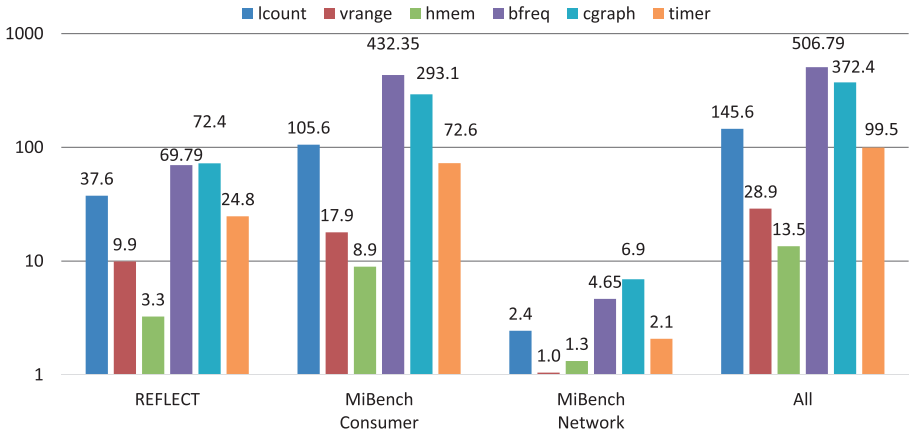


Figure 17. Aspectual bloat values taking into account the reuse of aspects.

DOI: 10.1002/spe

reused in different applications; (iii) multiple monitoring concerns described in LARA can be used in the same application code; and (iv) LARA is able to reduce the programming effort to support these monitoring concerns when compared with their manual implementation.

6. AN INDUSTRIAL CASE STUDY: ADVANCED INSTRUMENTATION AND HW/SW PARTITIONING

We now describe the use of the LARA aspect-oriented approach to identify code sections to be migrated to hardware accelerators in the context of two industrial applications. A typical first step in the assessment of the computational requirements includes the identification of the performance bottlenecks or computationally intensive code sections – also called *critical* sections. These code sections are commonly structured as *critical loop* constructs directly manipulating array variables.

To identify these code sections, developers make use of popular execution profiling tools such as *gprof*. Yet, such tools provide limited information about performance. In particular, they provide wall-clock time at the granularity of the function or procedure. This limits their usage in the first profiling stage, as various computationally intensive sections might be located inside the same function, or worse, a noncomputationally intensive code section can easily dilute or mask the performance weight of a section inside the same function. As a result, either the identification of critical code sections are missed or developers must restructure the code beforehand to isolate each potentially critical code section as a separate procedure, an arduous and error-prone process.

The approach described in this section relies on an automatic source-to-source transformation tool guided by LARA aspects, which has the added benefit of preserving the original source code while supporting customized performance metrics at the level of any C construct (such as loops) and irrespective of their function boundaries.

6.1. Methodology

In these experiments, we present the results of the integration of our aspect-oriented design flow with a commercial architecture developed by Coreworks [26]. This architecture consists of a RISC-type GPP, named FireWorks [26], coupled to a one-dimensional coarse-grained reconfigurable array (CGRA) coprocessing unit (on the basis of the one described in Ref. [27]), named SideWorks [26]. This architecture provides a master-slave execution model where the embedded GPP triggers the execution of a specific task in the CGRA by loading a configuration file and activating its execution by setting up specific registers in its I/O map.

As such, a key step in the development of efficient and high-performance implementation consists in the identification of the critical code sections, which are subsequently translated to a low-level description to configure/program the CGRA. To determine these code sections to be executed in the CGRA, the application source code must be profiled to identify hotspots in the code and instrumented to determine runtime information such as the number of times a loop has been executed or to count the average number of loop iterations.

As part of our evaluation, we wish to capture existing design practices and strategies devised by experts at Coreworks with LARA and automatically identify candidate code sections for hardware acceleration using our aspect-oriented design flow. Note that the actual process of translating candidate code sections to the Coreworks CGRA is not supported by our current source-level translation weaving process and would require a dedicated hardware compiler. Finally, we investigate how the manual and automated approaches compare in terms of the quality of the derived solution and their development effort.

6.2. Application concerns

6.2.1. Targeted benchmarks. In the evaluation and experiences described in this section, we focus on the performance analyses of two large applications from the audio encoding domain, namely, the MPEG and the G729 audio encoders. These encoders are representative of the typical type of applications used in Coreworks. These encoders consist of 15 files, 154 functions, and 9,022 lines of C code for MPEG and 33 files, 127 functions, and 11,105 lines of C code for G729. Despite the

DOI: 10.1002/spe

differences between them, both applications have characteristics that are common to the audio application domain, as well as to the digital image processing domain.

6.2.2. LARA strategies: instrumentation, execution, and partitioning. Our approach relies on a weaver, integrated into the Harmonic source-to-source transformation tool, to identify critical sections of the two audio applications mentioned earlier. As LARA allows to export an interface that captures runtime information such as number of executed iterations in a loop, these metrics can be subsequently folded into the selection criteria for transformation of a given loop.

In this evaluation, we consider two strategies, namely, strategies I and II, responsible to automatically select all critical outermost loops in a program that would benefit from hardware execution. Each strategy employs a specific heuristic (devised by the Coreworks experts) to select nested loops using static (source code) and dynamic (runtime extracted) parameters, as listed in Table VII. We identify nested loops by considering only their outermost loop body, with nested loops inside the outermost loop body not considered as part of the selection.

The key steps of these two strategies are the following:

- Step 1, instrumentation: The *profiling* aspect instruments the C code generating the 'woven code' to report the values of the dynamic parameters, when the modified application is executed.
- Step 2, execution: When the woven code is compiled and executed, the profiling functions inserted in the original source code report the values of the parameters of interest. The information retrieved at runtime is stored in an intermediate file called 'loop_analysis.js' that is processed in the next step by the *selection* aspect.
- Step 3, selection: The *selection* aspect (for hardware/software partitioning) chooses the critical loops on the basis of the criteria defined by the strategy. In this step, static and dynamic parameters (on the basis of runtime information collected from step 2) are used. The identification of the critical loops is based on a set of conditions, which classify all outermost loops as 'critical' or 'noncritical'.

The identification of the parameters and strategies has to be performed by an expert in the application and platform domains, in order to transfer his or her knowledge to the LARA language. Note, however, as we demonstrate in our evaluation, that the parameters and strategies can be reused for other applications in the same domain.

Dynamic parameters such as number of iterations and number of frames are obtained by the following aspects: (i) instrument the application in a first phase with self-monitoring code to obtain the required dynamic parameters and (ii) execute the instrumented code to generate dynamic parameters. Once the dynamic parameters are obtained using the aforementioned steps, loops are selected on the basis of specific criteria as described next.

Table VII. Parameters used by strategies I and II to identify critical loops.

Parameters	Description	Type	Strategy
`num_stmts`	Number of statements inside the block of nested loops	Static	I
`iterations`	Number of iterations of the innermost loop	Dynamic	I
`num_frames`	Number of audio frames to encode	Dynamic	I, II
`instances`	Number of times the block of nested loops is executed	Dynamic	I, II
`num_oper`	Number of operators inside an innermost loop	Static	II
`instances_avg`	Average number of times the block of nested loops is executed per frame	Dynamic	II
`iterations_avg`	Average number of iterations of the innermost loop	Dynamic	II
`num_ifs`	Number of 'if' statements inside the innermost loop	Static	II
`num_ifs_outer`	Number of 'if' statements inside the block of nested loops	Static	II
`if_count`	If innermost loop contains an 'if' statement, count the number of times the 'if' statement is executed	Dynamic	II
`then_count`	If innermost loop contains an 'if' statement, count the number of times the condition evaluates to true	Dynamic	II
`else_count`	If innermost loop contains an 'if' statement, count the number of times the condition evaluates to false	Dynamic	II

DOI: 10.1002/spe

Using the LARA approach, in step 3, each *critical* loop is encapsulated in a *#define* statement. This *#define* is used to toggle between the original code and a call to the SideWorks [26] coprocessor. Therefore, when compared with the manual profiling, the effort to perform the software/hardware partitioning is slightly reduced. The additional effort to define the SideWorks datapaths that implement the *critical* loop depends on the Coreworks design tools, but it is identical whether the LARA approach is used or not. We note that the use of other types of accelerators (e.g., custom hardware) would require a different translation process for the code sections to be migrated to the accelerator. For instance, when considering custom FPGA hardware, one might use a C-to-hardware tool to generate the accelerator's hardware. In case of the Coreworks design flow, the code sections must be manually translated to a native language that is able to program the coprocessor. The development of an automatic compilation tool that addresses this translation is out of the scope of this article.

Strategy I: This strategy employs four key parameters, as presented in Table VII, that are extracted from each candidate nested loop: number of loop body statements that we use to measure its complexity (*num_stmts*); total number of iterations (*iterations*) of the nested loop per instance; number of audio frames (*num_frames*); and number of instances that specifies the number of times the nested loop has been executed (*instances*). To acquire the number of iterations and the number of instances, the profiling aspect instruments each candidate nested loop: to count the total number of iterations per instance a counter is placed in the innermost loop body; to count the number of instances a counter is placed in the outermost loop body. Moreover, to derive the number of frames, the name of the variable storing the number of frames must be provided to the profiling aspect, which will automatically instrument the source code to monitor its value during the execution of the application. Once the instrumented application is executed, it will generate a report with the dynamic parameters for that particular execution.

Strategy I selects the outermost loop body with the following criteria:

```
select =  iterations * num_stmts > 512 OR (iterations * num_stmts >= 256 AND
          instances / num_frames >= 10)
```

Strategy II: This strategy improves strategy I by the following: (i) considering multiple executions of an application; (ii) taking into account conditionals; and (iii) supporting a better measure for code complexity. To support monitoring of multiple application executions, we compute the minimum, maximum and average number of loop instances (*instances_min*, *instances_max*, and *instances_avg*). We perform a similar statistical analysis for the number of loop iterations (*iterations_min*, *iterations_max*, and *iterations_avg*). To support conditionals, we included the computing of the frequency of *then* and *else* branches inside each innermost loop body (*then_count*, *else_count*). Finally, code complexity is measured by the number of arithmetic operators inside the innermost loop body (*num_oper*) with the possibility to customize the weights of different operators of a C expression. As with strategy I, we instrument the original source code to create a self-monitoring application that can compute the above dynamic parameters. Next, this woven application is executed, and a report with the runtime results is generated. This is followed by running the original application with the selection aspect, this time using the dynamic parameters computed in the previous step with static parameters to choose loops that are suited for hardware acceleration.

The loop selection criterion in Strategy II is now defined as follows:

```
select =  ((Condition1 OR Condition2) AND Condition3 AND Condition4 AND Condition5)
```

Specifically, a loop is selected if the expression above evaluates to true where the various conditions are described in Table VIII. Note that we can add another condition (i.e., condition 6), in which loops are rejected if they are enclosed in a function that executes less than a fraction of the total time of the application.

DOI: 10.1002/spe

Table VIII. Conditions for selecting loops using Strategy II based on parameters from Table VII.

Condition	Description
1	(a) innermost loop body with no IF constructs (b) number of operators in the loop are more than zero and less than a constant value (OPER_PER_BLOCK) num_ifs == 0 AND num_oper > 0 AND num_oper < OPER_PER_BLOCK
2	(a) innermost loop body contains one IF construct (b) one of the branches has a frequency greater than 95% (c) difference between total number of operators and the number of operators of the least frequent branch is less than OPER_PER_BLOCK num_ifs == 1 AND max(then_count / if_count, else_count /if_count) > 0.95 AND num_oper – least_freq_branch_num_ops < OPER_PER_BLOCK
3	reject the loop if contains 'if' statements outside the innermost: num_ifs_outer < = 1
4	selects outermost loops executed more than once: instances_avg > = 1
5	selects outermost loops with number of iterations greater than a specific value (MIN_ITERATIONS): iterations_avg > = MIN_ITERATIONS

6.3. Evaluation

6.3.1. Strategy comparison. The two strategies presented in the previous section have been derived from empirical observations and user expertise, and programed as LARA aspects. One important feature of our approach is that aspects are fully parameterized, and thus, these strategies can be easily changed in the context of automatic DSE and/or manual user experimentation. A second point is that strategy II is more complex than strategy I, taking into account more static and dynamic parameters providing more fine-grained control about the loop selection.

Table IX presents the results obtained by applying these two selection strategies, respectively, strategy I and strategy II, to the MPEG and G729 applications. More specifically, for each of the functions in these applications, we report the number of loops sections deemed as critical using the two selection strategies. For comparison purposes, we also include the results of using a manual selection approach relying on the profiling results generated by the *gprof* tool and by direct code inspection by developers. Regarding the 'manual' approach, we also rely on the use of various *printf* calls to capture relevant values of key parameters such as the number of loop iterations.

Table IX shows that strategy II excluded functions 'I_a_bit_allocation', 'ACELP_codebook64', and 'cor_h_cp' that contribute with 5.02%, 2.57%, and 2.04% of the MPEG global execution time, respectively. These functions are too complex to be mapped to the SideWorks CGRA, and this fact is correctly identified by strategy II but not by strategy I, as the latter only measures complexity by using the number of statements and the number of operators.

For both applications, all critical loops manually identified by developers were also automatically selected by the two LARA-based strategies. The LARA-based strategies, however, were able to identify additional critical loops for both applications, as are the examples of loops in the two FFT functions (*I_f_f_t* and *II_f_f_t* in MPEG code) and in the correlation function (*autocorr* in G729 code). A detailed analysis reveals that those loops do not belong to functions that emerge at the top of the profiling results, which were therefore not identified manually. The automatic identification of more suitable critical loops by the LARA strategies allows developers to converge more quickly to solutions that can achieve the required real-time performance.

In conclusion, the results automatically achieved by our LARA strategies provide information that cannot be easily derived manually or it can only be obtained at a high maintenance cost. Coreworks reported that an experienced developer required an entire day to manually identify the loops referred in the column "Manual approach" of Table IX. Conversely, the LARA technology performs this task automatically in less than 30 min. This effort, however, does not take into account the coding and testing of the LARA strategies, as these are reusable across other applications.

It is worth noting that the LARA strategy II refines the results obtained by LARA strategy I and selects fewer numbers of loops as critical. Although the migration to dedicated hardware of these loops may not degrade the performance, the effort required to manually map them to the SideWorks

DOI: 10.1002/spe

Table IX. Identification of the critical loops for the MPEG and G729 applications.

Application	Function	#Loops	% Global exec. time (profiling)	# Critical loops		
				Manual approach	LARA strategy I	LARA strategy II
MPEG	fft_gen	3	13.31	1	1	1
	filter_subband	6	38.16	3	3	3
	fft_ii_for2	1	3.37	1	1	1
	empty_buffer	2	0.29		1	1
	get_audio	6	0.18		1	1
	I_a_bit_allocation	7	5.02		1	
	I_subband_quantization	3	0.85		1	
	make_map	2	0.08		1	
	I_f_f_t	4	0.30		2	1
	I_hann_win	2	0.03		1	1
	I_tonal_label	5	0.12		1	
	I_minimum_mask	2	0.09		1	
	II_f_f_t	4	0.15			1
	II_hann_win	2	0.02			1
	Total	49	61.97	5	15	11
G729	syn_filte	4	4.73	1	1	1
	syn_filte_2	4	5.14	1	1	1
	residue_40_10	2	5.12	1	1	1
	autocorr	3	5.84	1	1	2
	lag_max1	3	7.69	1	1	1
	lag_max2	3	5.33	1	1	1
	lag_max3	3	2.66	1	1	1
	pre_process	1	0.6		1	
	ACELP_codebook64	33	2.57		8	
	cor_h_cp	12	2.04		5	
	cor_h_x	2	3.28		1	1
	convolve	2	6.19		1	1
	pred_lt_3	2	2.48		1	1
	lsp_pre_select	2	5.46		1	1
	lsp_select_1	3	0.77		1	1
	lsp_select_2	3	1.06		1	1
	Total	82	60.96	7	27	14

coprocessor is substantially reduced. Each loop with substantial complexity requires almost 1 day of work to be mapped to SideWorks as this mapping currently requires the translation from C code to the SideWorks low-level native programming language, the extraction of the loop from the application, and the integration of the communication and synchronization primitives in the application, and testing. An approximate figure of the effort reductions when using strategy II rather than strategy I for the MPEG and G729 applications would be around 4 and 13 days, respectively.

6.3.2. Productivity analysis. The profiling results described in the previous section and provided by LARA and Harmonic in this context have direct impact in the metrics defined by Coreworks. The following sentences summarize the conclusions of the senior software engineer from Coreworks who conducted this experimental evaluation:

- The development time is considerably reduced. The developer did not need to instrument the code manually with *printf* calls, which is a huge benefit for large applications, such as MPEG and G729. The experiments also show that the quality of the results obtained with the LARA approach is higher than the quality of results obtained manually. Critical loops not identified have a negative impact in the performance of the overall system in terms of execution time and energy consumption. In these case studies, we observed significant productivity improvements considering the time required by an expert Coreworks developer to analyze and select the loops without using LARA and the time using the LARA approach. More specifically,

DOI: 10.1002/spe

for large applications, the profiling takes approximately one working day. Using the LARA approach, the profiling time is reduced by one order of magnitude.
- A major advantage of the LARA approach lies in the ability to reuse the same aspects (LARA files) across applications and possibly by nonexpert developers about the target architecture. In this particular case, the LARA aspects used to identify the critical loops were reused in both applications without any modification. This reusability represents a huge benefit to development time, as well as to nonrecurring engineering costs.
- LARA allows powerful ways to specify sophisticated strategies (possibly reusable) for monitoring, instrumentation, and for customized profiling schemes.
- One of the LARA approach strengths is the flexibility to express and customize criteria for selecting loops that may help developers and/or other tools to decide about the mapping of those loops.
- The use of LARA allowed to automatically identify critical and SideWorks suitable loops in large applications (MPEG encoder and G729).
- The LARA aspect-based approach truly helps and enhances the methodology used by Coreworks on loop analysis and loop selection, allowing them to experiment with different strategies by revising the heuristics.

We performed a quantitative evaluation using specific AOP metrics with strategy II (which is an enhancement of strategy I) and with three applications, namely, MPEG, G729, and a universal audio encoder, which merges the first two encoders and selects the encoding format according to dynamic requirements. Table X presents some of the characteristics of the LARA code for strategy II and of the application source code used in the experiments. Table XI presents the metrics related to the application of the strategy and to the resultant C code. The LARA code for strategy I consists in three aspect definitions (three source files): (i) *loop_analysis*; (ii) *loop_selection*; and (iii) *loop_sideworks*, with 108, 156, and 46 LOC (a total of 310 lines of LARA code), respectively.

For the 'loop_analysis' aspect, the weaving needed to consider a large number of join point shadows. In this example, the value for CDLOC, tangling ratio, and aspectual bloat are high and clearly show the huge efforts needed if the code insertions were carried out manually. In fact, the weaving time for the examples illustrates the substantial code analysis and code insertions needed. The aspectual bloat for reuse is 85.11 and 6.74 for aspects 1 and 3, which, with only three applications, already show high advantages of reuse in terms of code.

7. RELATED WORK

The use of programming approaches for code instrumentation, compiler strategies, and mapping decisions has been addressed by various authors. In this section, we highlight related work in the areas of AOP and specification of strategies for compilers.

Table X. Some characteristics of the LARA aspects considering Coreworks strategy II and application C code.

	LARA aspects			Application (C source code)								
				Universal audio encoder			MPEG			G729		
Aspect	# defs	# sources	# LOC	# files	# funcs	#LOC initial	# files	# funcs	#LOC initial	# files	# funcs	#LOC initial
loop_analysis (1)	1	1	108	42	213	16,880	9	88	6,102	34	131	11,078
loop_selection (2)	1	1	156									
loop_sideworks (3)	1	1	46									

LOC, lines of code.

DOI: 10.1002/spe

Table XI. Metrics considering Coreworks strategy II and the three applications.

Application	Aspect	#jp selected	#jp inserted	% funcs modified (insert coverage)	#LOC final	CILOC	CDLOC	Tangling ratio (%)	Aspectual bloat	Woven time
Universal audio	1	13,374	990	62.91	21,449	1.27	1954	9.09	42.77	30 m
encoder	2	609	—	—	—	—	—	—	—	15 s
	3	651	78	13.15	17,035	1.01	156	0.92	3.37	
MPEG	1	5029	219	39.77	7,109	1.27	438	6.16	9.32	6 m 49 s
	2	217	—	—	—	—	—	—	—	
	3	226	25	10.23	6151	1.01	56	0.91	1.07	
G729	1	8,849	774	76.34	14,694	1.27	1526	10.39	33.48	15 m
	2	393	—	—	—	—	—	—	—	52 s
	3	427	53	14.50	11,184	1.01	104	0.93	2.30	

LOC, lines of code; CILOC, concern impact lines of code; CDLOC, concern diffusion lines of code.

DOI: 10.1002/spe

7.1. Aspect-oriented programming

Aspect-oriented programming has been the subject of intense research over the last decade. AspectJ [7, 19, 20], one of the most widely known AOP languages, is an AOP extension to Java aimed at providing better modularity for Java programs. AspectJ contributes to cleaner and better code by modularizing programs, providing solution for several crosscutting concerns such as monitoring, logging, debugging, synchronization, and performance. Another example with noticeable success is AspectC++ [28, 29], 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. LoopsAJ [30] 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 [31] is a refinement of AspectJ that allows the specification (using labels) of join points at the level of individual statements such as *if* and *while* loops. Rajan and Sullivan [32] propose Eos-T, a version of the aspect-oriented language Eos [33]. Both Eos and Eos-T extend C# with AOP concepts to include branches and loops as join points. We have extended in LARA their notion of *advices* to include code transformations, compiler optimizations, mapping options, and property specifications, report generation, besides code weaving.

Reflecting AOP's growing acceptance, several AOP extensions have been proposed to other languages and domains of applicability. For instance, AspectMatlab [34] 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*).

While LARA has been inspired by many AOP approaches, including AspectJ and AspectC++, it differs from these efforts in several ways. First, we extend the capabilities associated with types, such as their shapes and precision (as described in our previous work [35, 36]). In addition, the join point model allows the specification of pointcuts in virtually all points in a program avoiding or diminishing the need for labels, annotations, and pragmas. Second, and while we have opted mechanisms similar to the ones used by AOP approaches based on functional queries [37], in LARA, pointcut expressions allowed by select sections of the aspect modules are able to define composable select expressions (similar to composable queries) as in Ref. [37]. 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. Third, we have defined a 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 [9]. On the basis of 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 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 [38, 39]. 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. 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 [40]). As an example, Catapult-C [41] supports 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 syntactic and semantic join points and join points exposed along compiler sequences.

DOI: 10.1002/spe

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*. [42] in the context of software product lines. By exposing to aspects the characteristics of the target architecture, we promote toolchain adaptability for different architectures. Note that besides code variations we also support AOP-based strategies that allow different implementations by controlling key toolchain stages.

The use of a join point model to process LARA aspects into an Aspect-IR representation allows LARA to address different design flows and different host programming languages (our core experiences have been focused on C, but we had experiments with LARA aspects for MATLAB). Relevant to our approach are the join point and mapping models proposed by Jackson *et al*. [43] to achieve a cross-language AOP in the context of the .NET framework. They envision a programming language-independent AOP approach through an XML AOP language based on AspectJ. Although regarding the flexibility to deal with more than one host programming language, common points with our approach are seen, they rely to the AOP approach of AspectJ that is not suitable for our objectives.

Other approaches address language-independent tools [44] for weaving existing components with aspects written in the language of choice. Others target application components in different languages and use a common representation of the components to apply the aspects (e.g., UniAspect [45]). UniAspect keeps a similar syntax to AspectJ and introduces "@" annotations for identifying target language. This is similar to LARA as the insert statements can also identify the target language.

The LARA support to access to a particular join point and to its attributes (including the meta-attributes) has conceptual similarities to the *ThisJoinPoint* variable used in AspectJ [19, 20] and AspectC++ [28, 29]. For example, in AspectJ, the *thisJoinPoint* variable contains meta-information about the join point (e.g., the method signatures and arguments when in the presence of a method call). In our case, each of the join point types in the join point model has a number of attributes. Our approach allows the use of specific information obtained at the moment by static evaluation or by runtime information feedback, for example, during execution or profiling of the application.

7.2. Controlling compilation

The PATUS framework [46] defines a domain-specific language specifically geared toward stencil computations. It allows programmers to define a compilation strategy for automated parallel code generation using both classic loop-level transformations such as loop unrolling and loop splitting, as well as exploiting architecture-specific extensions such as SSE instructions. To increase the flexibility and performance portability of the generated codes, PATUS generates parameterized stencil codes, which then interface with an autotuner for specific parameter value selection.

The PATUS framework differs substantially from the LARA framework both in scope and expressive power. Whereas PATUS focuses exclusively on the specific notions in stencil domains, and on its operators, LARA is a domain-specific language geared toward arbitrary computations. Rather than relying on specific abstractions for a given domain, LARA leverages the transformational power of third-party engines. In addition, the join point model exported by LARA allows users to develop very sophisticated transformation and mapping strategies clearly beyond the reach of the PATUS approach given its limited set of abstractions.

Existing high-end compilers do not completely expose the set of transformations but rather only allowed a limited set of them to be controlled by developers via pragma annotations. Research efforts, such as CHiLL [47] and POET [48], have attempted to expose source-level transformations in a controlled fashion by offering a script specification of the sequence of transformations and corresponding parameters.

These analyses and transformation frameworks also differ from the LARA-based approach in various respects. They have concentrated exclusively on source code transformations for either scientific computing or, as is the case of POET, on multilanguage translation focusing on the important

DOI: 10.1002/spe

issue of performance and autotuning for performance portability. Instead, the LARA-based approach bridges the gap between hardware synthesis and software compilation. In addition to directing transformation engines, LARA allows programmers to customize, in a nonintrusive way, the source code to be used by other tools. Uses of LARA regarding code transformations and compiler optimizations as the main actions are presented in Refs. [9, 10].

In the context of DSE, there have been approaches to enable developers to customize the composition and parameterization of design transformations through scripting, in order to automatically derive designs that can meet specific goals (e.g., Ref. [49]). LARA complements DSE approaches by providing a unifying DSE platform, which captures and enacts evolving strategies with full design flow control [9]. As shown in Ref. [11], LARA can be used to specify sophisticated DSE strategies involving all the stages of a design flow.

8. CONCLUSION

This article presented a novel AOP language named LARA, which provides separation of concerns, including NFRs and strategies, for the mapping of high-level applications to high-performance heterogeneous embedded systems. We described how LARA supports code instrumentation, code insertion, profiling customization, and selection of critical code sections for hardware acceleration, all in the context of development for embedded computing systems. Our LARA-based tools have been evaluated with real-life industrial C applications and the experimental results provide strong evidence of its usefulness and practical benefits in terms of application and performance portability and programmer productivity and portability across distinct target 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. Ongoing work is focusing on the dynamic weaving support for the LARA technology and on the traceability and dependability aspects for specifying sequences of code transformations and compiler optimizations. Future work will address a meta-model for LARA aspects that will be mapped to join point and attribute models of different host languages, to allow aspects to be reused across multiple software languages (e.g., C and MATLAB).

A number of demos providing the use of LARA to control/guide different compilers are available online: MATISSE[¶] (a MATLAB to C compiler), Reflectc[|] (a C compiler), and MANET[**] (a tool for source-to-source transformations).

ACKNOWLEDGEMENTS

This work was partially supported by the European Community's Framework Program 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. T. Carvalho also acknowledges the Portuguese Science Foundation (FCT) support through PhD grant SFRH/BD/90507/2012. J. G. F. Coutinho and W. Luk acknowledge the partial support by the UK EPSRC and by FP7 projects nos. 257906, 287804, and 318521.

REFERENCES

1. Hauck S, DeHon A. Reconfigurable computing: the theory and practice of FPGA-based computation. Morgan Kaufmann, Nov. 2007.
2. Cardoso JMP, Diniz P, Weinhardt M. Compiling for reconfigurable computing: a survey. *ACM Computing Surveys (CSUR)* 2010; **42**(4), Article 13:1–65.
3. Asher YB, Rotem N. Using memory profile analysis for automatic synthesis of pointers code. *ACM Transactions on Embedded Computing Systems* 2013; **12** (3), Article 68:21.

[¶] http://specs.fe.up.pt/tools/matisse/
[|] http://specs.fe.up.pt/tools/reflectc/
[**] http://specs.fe.up.pt/tools/manet/

DOI: 10.1002/spe

4. Holewinski J, *et al*., Dynamic trace-based analysis of vectorization potential of applications. in *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, SIGPLAN Not. 47, 6 (June 2012), ACM New York, NY, USA, pp. 371–382.
5. Lazarescu MT, Lavagno L. Dynamic trace-based data dependency analysis for parallelization of C programs. in *Proc. 12th IEEE Int'l Working Conference on Source Code Analysis and Manipulation*, pages 126–131, Sept. 2012.
6. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M. Irwin J. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP'97)*, Springer-Verlag, LNCS 1241, June 1997, pp. 220–242.
7. Kiczales G, Hugunin J, Hilsdale E, Kersten M, Palm J, Lopes C, Griswold B, Isberg W. Aspect-oriented programming. Final Technical Report, AFRL-IF-RS-TR-2003-173, Palo Alto Research Center, USA, July 2003.
8. Cardoso JMP, *et al*. "LARA: An Aspect-Oriented Programming Language for Embedded Systems," in *Proc. 11th Annual Int'l Conference on Aspect-Oriented Software Development (AOSD'12)*, Potsdam, Germany. pp. 179-190.
9. Cardoso JMP, Diniz P, Coutinho JG, Petrov Z (eds.), *Compilation and synthesis for embedded reconfigurable systems: an aspect-oriented approach*, 1st edition. Springer: New York, NY, USA, 2013.
10. Cardoso JMP, *et al*. Specifying compiler strategies for FPGA-based systems. in *Proc. 20th Annual IEEE Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM'12)*, Toronto, Ontario, Canada, April 29-May 1, 2012, pp. 192–199.
11. Cardoso JMP, *et al*. Controlling a complete hardware synthesis toolchain with LARA aspects. *Elsevier Journal on Microp. and Microsystems* 2013; **37**(8):1073–1089.
12. Filman R, Friedman D. Aspect-oriented programming is quantification and obliviousness. In *Chapter 2 of Aspect-oriented Software Development*, Filman RE, Elrad T, Clarke S, Akşit M (eds.), Addison-Wesley Professional, 2004; 21–36.
13. Masuhara H, Kiczales G, Dutchyn C. A compilation and optimization model for aspect-oriented programs. In *Proceedings 12th Int'l Conf. on Compiler construction (CC'03)*, G. Hedin (*ed*.). Springer-Verlag, Berlin, Heidelberg, pp. 46–60.
14. Myers CG, Baniassad ELA. Metaproperty aspects. in *Proc. 8th ACM Int'l Conf. on Aspect-oriented Software Development (AOSD'09)*, Charlottesville, Virginia, USA. 2009, pp. 231–242.
15. Bispo J, *et al*. The MATISSE MATLAB Compiler – a MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms. in *IEEE Int'l Conf. on Industrial Informatics (INDIN'2013)*, Bochum, Germany, 29-31 July 2013, pp. 602–608.
16. Cardoso JMP, Coutinho JGF, Carvalho T. LARA programming language specification, v2.0. *REFLECT Internal Technical Report*, Sept. 2012.
17. Flanagan D. JavaScript: the definitive guide. O'Reilly Media; 6th edition (May 10, 2011).
18. ECMA-262: ECMAScript language specification, 5.1 Ed., June 2011, © Ecma International 2011, http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
19. Kiczales G, *et al*. An overview of AspectJ. *In Proce. 15th European Conf. on Object-Oriented Programming (ECOOP'01)*, Springer-Verlag, London, UK, UK, 2001, pp. 327–353.
20. Gradecki J, Lesiecki N. *Mastering AspectJ: Aspect-oriented Programming in Java*. J. Wiley & Sons, Inc.: New York, NY, USA, 2003.
21. Wand M, Kiczales G, Dutchyn C. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems* 2004; **26**(5):890–910.
22. Luk W, *et al*. A high-level compilation toolchain for heterogeneous systems. in *Proc. IEEE Int'l SOC Conf. (SOCC'09)*, Sept. 2009, pp. 9–18.
23. Figueiredo E, *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.
24. Lopes CV. D: a language framework for distributed programming. PhD thesis, College of Computer Science, Northeastern University, Nov. 1997.
25. Guthaus MR, *et al*. MiBench: a free, commercially representative embedded benchmark suite. In *Proc. IEEE Int'l Workshop on Workload Characterization (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 2001, pp. 3–14.
26. Coreworks SA, http://www.coreworks-sa.com/
27. De Sousa JT, *et al*. Reconfigurable coprocessor architecture template for nested loops and programming tool. Coreworks Sept. 25 2012, United States Patent 08276120.
28. Lohmann D, Spinczyk O. *Aspect-oriented programming with C++ and AspectC++*, tutorial *at 6th Int'l conf. on aspect-oriented software development (AOSD'2007)*. Vancouver, British Columbia, Canada, March 12-16, 2007.
29. Spinczyk O, Gal A, Schröder-Preikschat W. 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, pp.* 53–60.
30. Harbulot B, Gurd JR. 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.
31. Poggi M. @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.
32. Rajan H, Sullivan K. Aspect language features for concern coverage profiling. in *Proc. 4th Int'l Conf. on Aspect-Oriented Software Development (AOSD'05)*, ACM, New York, USA: Chicago, Illinois, 2005, pp. 181–191.

DOI: 10.1002/spe

33. Rajan H, Sullivan K. Eos: instance-level aspects for integrated system design. *SIGSOFT Software Engr. Notes* 2003; **28**(5):297–306.
34. Aslam T, Doherty J, Dubrau A, Hendren L. AspectMatlab: an aspect-oriented scientific programming language. in *Proc. 9th Int'l Conf. on Aspect-Oriented Software Development (AOSD'10)*. ACM, New York, NY, USA, 2010, pp. 181–192.
35. Cardoso JMP, Fernandes J, Monteiro M. *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.
36. Cardoso JMP, *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.
37. Eichberg M, Mezini M, Ostermann K. Pointcuts as functional queries. In *Programming Languages and Systems*, Chin W-N (ed.). Springer: Berlin, Heidelberg, 2004; 366–381.
38. Alves V, *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.
39. Adams B, Meuter W, Tromp H, Hassan A. Can we refactor conditional compilation into aspects? in *Proc. 8th ACM Int'l Conf. on Aspect-Oriented Soft. Development (AOSD'09)*, 2009, pp. 243–254.
40. Ferrer R, *et al*. Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. in *Proc. LCPC*, 2010, pp. 215–229.
41. © Mentor Graphics. Catapult C synthesis, http://www.mentor.com/esl/catapult
42. Alves V, *et al*. Extracting and evolving code in product lines with aspect-oriented programming. In *Trans. on Aspect-oriented Software Development IV*, Rashid A, Aksit M (eds). Springer: Berlin, Heidelberg, 2007; 117–142.
43. Jackson A, Clarke S. SourceWeave.NET: cross-language aspect-oriented programming. in *3rd Int'l Conf. on Generative Programming and Component Engineering (GPCE'04)*, Vancouver, Canada, Oct. 24-28, 2004. Lecture Notes in Computer Science (LNCS), Vol. 3286, Springer, 2004, pp. 115–135.
44. Lafferty D, V Cahill. Language-independent aspect-oriented programming. In *Proc. of the 18th annual ACM SIGPLAN Conf. on Object-oriented programing, systems, languages, and applications (OOPSLA '03)*. ACM, New York, NY, USA, pp. 1–12.
45. Ohashi A, *et al*. UniAspect: a language-independent aspect-oriented programming framework. In *Proc. Workshop on Modularity in Systems Software (MISS'12)*. ACM, New York, NY, USA, 2012, pp. 39–44.
46. Christen M, Schenk O, Burkhart H. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. in *Proc. IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS'11)*, IEEE Computer Society, 2011, pp. 676–687.
47. Hall M, Chame J, Chen C, Shin J, Rudy G, Khan M. Loop transformation recipes for code generation and autotuning. in *Proc. 22nd Int'l Conf. on Languages and Compilers for Parallel Computing (LCPC'09)*, Newark, DE, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 50–64.
48. Yi Q. POET: a scripting language for applying parameterized source-to-source program transformations. in *Software Practice and Experience*, John Wiley & Sons, Inc., 42(6), June 2012, pp. 675–706.
49. Liu Q, Todman T, Coutinho J, Luk W, Contantinides G. 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.

DOI: 10.1002/spe