PAPER

A Ruby-Based Hardware/Software Co-Design Environment with Functional Reactive Programming: Mulvery

Daichi TERUYA[†a)], ***Student Member and*** **Hironori NAKAJO**[††], ***Member***

SUMMARY Computation methods using custom circuits are frequently employed to improve the throughput and power efficiency of computing systems. Hardware development, however, can incur significant development costs because designs at the register-transfer level (RTL) with a hardware description language (HDL) are time-consuming. This paper proposes a hardware and software co-design environment, named *Mulvery*, which is designed for non-professional hardware designer We focus on the similarities between functional reactive programming (FRP) and dataflow in computation. This study provides an idea to design hardware with a dynamic typing language, such as Ruby, using FRP and provides the proof-of-concept of the method. Mulvery, which is a hardware and software co-design tool based on our method, reduces development costs. Mulvery exhibited high performance compared with software processing techniques not equipped with hardware knowledge. According to the experiment, the method allows us to design hardware without degradation of performance. The sample application applied a Laplacian filter to an image with a size of 128×128 and processed a convolution operation within one clock.

key words: *SoC FPGA, HLS, functional reactive programming*

1. Introduction

Due to the end of Moore's Law, making improvements to software performance are reaching a limit. Therefore, computation using custom hardware (circuits) is frequently hired to improve the throughput, latency, and power efficiency of computer systems. In this situation, reconfigurable devices, such as a field-programmable gate array (FPGA), is attractive from both academic and industrial perspectives. An FPGA is a type of very-large-scale integration (VLSI) device that an end-user can reconfigure to build custom digital circuits with reduced manufacturing costs.

1.1 Problems with Designing Hardware

Hardware development, however, can lead to significant development costs due to difficulties in describing the register-transfer level (RTL) with a hardware description language (HDL). HDL often tends to cause bugs and certain challenges related to debugging. In contrast, software-development is characterized by mature debugging techniques and analysis methods, in addition to well-developed coding techniques. Thus, software has relatively low development costs and high productivity. Additionally, more tools for software development are available than for hardware development, which are often free to use. Therefore, FPGA development incurs higher development costs than software development, such that FPGA applications are limited despite their benefits in terms of energy and performance.

To solve this problem, there are two well-known approaches. The first is meta-programming. Tools of this type generate HDL descriptions from code written in a domain-specific language (DLS) or macros with existing programming languages. The amount of hardware description can be less than when using an HDL, such that the abstraction level is equivalent to the HDL descriptions. Such tools require knowledge for designing hardware even though the tool based on a programming language. Hence, the techniques do not reduce design costs but only development costs.

The second approach is the adoption of high-level synthesis (HLS) techniques that synthesize HDL descriptions from behavioral descriptions written in a programming language, such as C/C++ or Java. There are currently HLS tools based on various languages [1], [2]. Moreover, FPGA giants, such as Xilinx and Altera/Intel, have released HLS tools based on C/C++ language [3], [4]. Typically, HLS tools are specialized for software acceleration. However, users must add certain annotations to the code to optimize the performance. HLS tools require knowledge of specific techniques to analyze the generated circuit and put effective annotation to the source code so that a room remains to reduce design and development costs. The central cause of this problem is a difference in the programming paradigm among programming language and HDL. Programming languages rest on control-flow style, but HDL rest on data-flow style. This fact leads to two problems. Firstly, since software programs have no information for parallelism, HLS tools require annotations similar to parallel computing programs. Secondly, Because there is no information for dataflow in software programs, we need to control that which piece of the code does the tool synthesizes into a module, or the generated circuit might be unnecessarily either huge or low performance.

Furthermore, co-design environments for system-on-chip (SoC) FPGAs, which have microcontroller units (MCUs) as the hard-macro on one chip, are widely used. However, as these environments are also made to accelerate

Manuscript received August 29, 2019.
Manuscript revised February 12, 2020.
Manuscript publicized May 22, 2020.

[†]The author is with Graduate School of Engineering, Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan.
[††]The author is with Institute of Engineering, Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan.
a) E-mail: teruya@nj.cs.tuat.ac.jp

DOI: 10.1587/transinf.2019EDP7233

software, opportunities for their use are also limited. Therefore, there is a lack of descriptiveness when describing the communication among hardware parts (or both hardware and software parts). As such, our second goal is to propose a new co-design tool of hardware and software. Our method achieves flexible software and hardware cooperation and minifies of adding extra-codes in user codes to realize the cooperation.

For rapid development, software engineering occasionally adopts a lightweight language (LL) [6], such as Python or Ruby. Contrarily, in hardware development, there is room for improvement because, to the best of our knowledge, fewer frameworks that focus on development efficiency exist than software development.

1.2 Our Approach with Functional Reactive Programming

For these reasons, this study proposes a new hardware design tool, named as *Mulvery framework*. The principal aim of this study is to generate efficient circuits without adding annotations by hand and any extensions of programming language. This paper proposes the concept of our novel hardware design method. Mulvery introduces the concept of functional reactive programming (FRP) [8], [9] which is a programming paradigm similar to hardware design. Mulvery generates hardware from a Ruby program written using a pre-defined operator set provided by Mulvery library. In this study, we adopted ReactiveX (Rx) [10] as a basic implementation of FRP. Rx is a widely used standard operator-set for development, such as web programming and Android OS applications. Mulvery then translates each operator-chain in an input program to a pipeline of hardware modules written in HDL. The most operators are high-order functions and take lambda abstractions as arguments so that the operators are able to handle any type of data.

As Rx is an API set defined for software programming, Mulvery is similar to an HLS tool. Since FRP is a type of data-flow style programming, Mulvery can converts a program into a data flow graph (DFG) directly. Hence, Mulvery is not an HLS tool but, rather, it is one of a meta-programming tool for HDL.

The principal aim of this study is to design circuits without adding annotations by hand and any extensions of programming language. This paper provides a proof-of-concept of our novel hardware design method. To summarize briefly, this paper provides the following contributions:

- Reduce hardware design costs with a lightweight language and FRP (Sect. 2.1),
- Automated co-design of software and hardware (Sect. 2.2), and
- Provide hardware design techniques using dynamic typing languages (Sect. 5).

The remainder of this paper is organized as follows. Section 2 explains the Mulvery framework. Section 3 discusses relevant previous studies and Sect. 4 provides an overview of FRP. Section 5 presents a method to generate HDL codes using FRP. Mulvery is evaluated in Sect. 6 and Sect. 7 provides a summary and conclusion of our study.

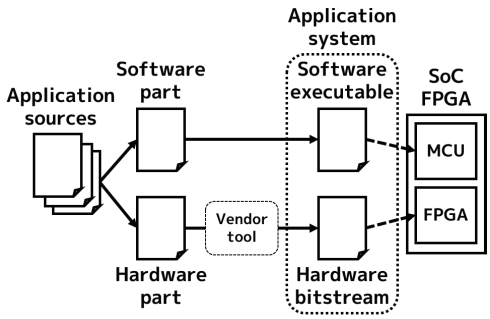

Fig. 1 The synthesizing flow of Mulvery.

2. Mulvery Framework

We have developed a hardware and software co-design framework, Mulvery. This section shows the structure of our Mulvery framework and describes how users can develop an application with Mulvery.

Mulvery targets platforms using an SoC FPGA, which includes both MCUs and FPGA logic, such as the Zynq-7000 series released by Xilinx, Inc. An application generated by Mulvery must have both an executable file for the MCU and a bit file for the FPGA. Therefore, Mulvery separates an input program into software and hardware components and automatically adds codes and circuits, such that both MCU and FPGA can collaborate and simultaneously work. Figure 1 shows a generation flow of an application for an SoC FPGA using Mulvery.

2.1 Hardware Design with a Dynamic Typing Language

Compilers cannot synthesize a description written with a dynamic typing language into a circuit since compilers are unable to determine the types of variables with static syntactic analysis in a dynamic typing language. Mulvery uses one of FRP's characteristics that an interpreter can dynamically analyze a program written with FRP without any sample input data of the application and can build a dataflow graph. In Mulvery library, codes that generate a corresponding node of its dataflow graph are implemented instead of the actual functionality. Hence, although *map*, *scan*, and *subscribe* method in Fig. 8 are executed when Mulvery is generating the hardware, the Ruby interpreter generates dataflow graph instead of actual calculations. Section 5 discusses details on how to generate HDL from a program. Although the use of a dynamic typing language has not been sufficiently studied as a hardware design tool, we suggest that this study can offer a significant contribution to hardware design methodologies. Our proposal can be applied not only with Ruby but also with various programming languages, regardless of whether its type system is dynamic or static.

```
1  class MySystem < Mv::MulveryBase
2    def build_dataflow()
3      ...
4    end
5
6    def main()
7      ...
8    end
9  end
```

Fig. 2 A class inheriting *MulveryBase*.

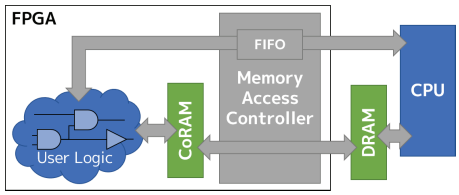

Fig. 3 The basic architecture of a circuit generated using Mulvery.

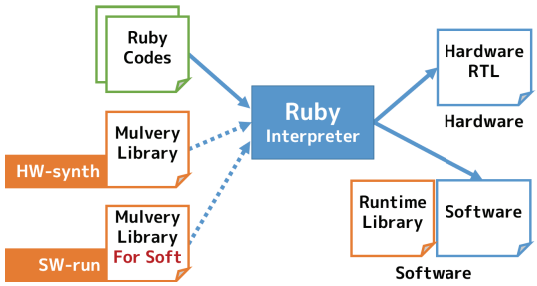

Fig. 4 The structure of the Mulvery framework.

```
1  def build_dataflow()
2    data = (100..110).to_a()
3
4    _numbers = Mv::Observable.from_array(data)
5    _timer = Mv::Observable.interval(1)
6    @numbers = \\
7        Mv::Observable.zip(timer, _numbers)
8    @even_numbers = \\
9        @numbers.select(){ |t, n| n % 2 == 0 }
10 end
```

Fig. 5 An example of the *build_dataflow* method.

2.2 Co-Design of the Hardware and Software

If a part of an application is too complicated to code with Mulvery's operators, typically, the part is not suitable to be an FPGA-based accelerator, even if either using another HLS tool; such a part needs hardware experts to be improved the performance. Hence, programmers should reorganize the algorithm to rewrite into a dataflow-style to code with Mulvery's operators. If it is hard to realize, the part is suitable to be calculated on an MCU.

Thus, Mulvery retains the components that are not Mulvery's operators as "software" and translates only the Mulvery's operators into hardware. Behavior that is easy to describe in a dataflow model, such as an Rx description, is suitable for processing with hardware. In contrast, communication processing, such as a TCP/IP protocol stack, is adequate for software because an FPGA implementation for such a transaction consumes hardware resources and degrades system performance. Therefore, such parts should be implemented as software from a performance and productivity perspective.

Typically, for most HLS tools, users must modify and optimize applications to improve performance. In contrast, Mulvery offers increased performance without tuning. Mulvery does not synthesize parts of a program that expect performance degradation if it is offloaded to FPGA; such a part runs on MCU. A Muvery program express temporal and spatial parallelism.

(1) Programming model

A part of program written with Mulvery's operators is explicitly separated for simple synthesis in the Mulvery framework. Specifically, the MulveryBase class shown in Fig. 2 is used as the top-level of a target application. Mulvery synthesizes a circuit from Mulvery's operators written in the *build_dataflow* method. The main method is executed on an MCU as the software component. The software component transmits data and passes results to and from the FPGA via instance variables defined in its *build_dataflow* method. To understand this, let us view a sample code in the following subsection.

(2) Data sharing among hardware and software

Figure 3 shows the basic architecture of a circuit generated using Mulvery. The CoRAM architecture [11] is used to communicate among modules on an FPGA and CPU(s). CoRAM is one of shared memory architectures designed to communicate among multiple hardware modules on reconfigurable devices. When transferring an event from a module on an FPGA to a CPU, the module stores data from the event in a CoRAM assigned to the module. Therefore, software components can read data from the hardware components, similar to memory-mapped I/O.

2.3 Developing an Application with Mulvery

Figure 4 shows the structure of the Mulvery framework. To use Mulvery, the application program requires the *Mulvery Library*. As the Mulvery framework does not extend the syntax of Ruby and is built as a Ruby library, Mulvery can be used on a regular Ruby interpreter without a dedicated compiler or preprocessor. This following subsection explains the workflow to develop an application with Ruby and Mulvery.

(1) Defining dataflows

Codes written with Mulvery's operators must be placed in the *build_dataflow* method of the *MulveryBase* class, and it will be the dataflow machine in an FPGA.

Figure 5 shows an example of the *build_dataflow*

```
def main()
  @numbers.subscribe() do |e|
    print("numbers : #{e}")
  end
  @even_numbers.subscribe() do |e|
    print("even_numbers : #{e}")
  end
  sleep(10) # observe events while 10 seconds
end
```

```
numbers : [0, 100]
even_numbers : [0, 100]
numbers : [1, 101]
numbers : [2, 102]
even_numbers : [2, 102]
numbers : [3, 103]
numbers : [4, 104]
even_numbers : [4, 104]
...
```

Fig. 6 An example of the *main* method and its output.

method. In Fig. 5, a dataflow is defined using a timer module and an array of certain numbers. The *_number* observable is an event stream, where each event contains a number, such that the *_timer* generates an empty event every second. The *zip* operator waits for an event from two observables. When it receives an event from each observable, the operator zips these events into an object and emits it as a new event. In short, the *@numbers* observable takes data from *data* and emits the data as an event every second.

A *select* operator filters items that does not pass a test specified by the lambda abstraction. Thus, *@even_numbers* observable emits an even number as an event every two seconds. Mulvery provides multiple ways to generate an observable (e.g., *from_array, from_IO, and timer*).

(2) Defining a software component

A software component is specified in the *main* method of the *MulveryBase*. The software component subscribes data from hardware components via observables defined in *define_dataflow* method as instance variables. Subscribers of each observable are registered with the *subscribe* method. Figure 6 shows an example of a *main* method implementation and its output. In the sample code, the subscribers merely display the data when it is observed.

(3) Behavioral test

As simulations of RTL are time-consuming, behavioral tests is a better way to ensure that the behavior of a program is correct while coding. Applications can use the "Mulvery Library for Soft" instead of the "Mulvery Library" for behavioral tests. When an application loads the test library, the program runs as software, without hardware.

3. Related Works

3.1 Hardware Design Tools

PyMTL [12] is a custom-hardware design tool based on Python, which can be referred to as a multi-paradigm design tool because it contains three levels of abstraction: the function level (FL), clock level (CL), and register-transfer level (RTL). As FL descriptions are a type of dataflow programming, PyMTL is similar to Mulvery. However, PyMTL users must implement each function in CL or RTL. Therefore, PyMTL is not designed for software engineers.

Chisel [13] and CλaSH are hardware design tools based on a functional programming language. Chisel is currently used as a softcore implementation of RISC-V [14] and the RISC-V implementation is published as open-source hardware. Although these tools may decrease implementation and debugging costs, design costs are still high because they are not behavioral descriptions.

Veriloggen [15] is also a Python-based multi-paradigm Verilog HDL code constructor. The programming model for Veriloggen is sufficiently abstracted from the Verilog HDL code as an AST for secure handling of hardware specifications in Python. Pyverilog provides several high-level operations (e.g., the Reduce instruction to data streams). However, input and output data types for the operations are limited and such operations require extra codes to fit the data types before and after the operation. Thus, there is room to improve productivity.

3.2 Hardware and Software Co-Design Tools

FPGA vendors have released this type of tool to encourage hardware acceleration of software (e.g., Xilinx SDAccel [16] and Intel FPGA SDK for OpenCL [3]). Typically, to offload parts of the process to an FPGA, users must select and annotate target functions. Users must analyze and tune the target software to determine the parts that will be offloaded. These tools are peculiarly suited to hardware engineers since knowledge of the hardware is required to perform this task.

There are several tools that automatically select the hardware parts. However, there are a several tools that automatically select the hardware parts. Warp Processing [17] analyzes a program that is running on an MIPS processor and dynamically offloads slow parts. Thus, users do nothing to optimize the system but must wait for trial-and-error analysis, which is performed by the tool. LegUp [1] can statically construct a co-designed system. However, LegUp currently only supports a pure hardware architecture and is, therefore, not a co-design tool. Consequently, methodologies for automatic architecture exploration in co-design environments further studies are needed.

4. Functional Reactive Programming

This study focuses on FRP to realize a novel hardware and software co-design environment. FRP is a paradigm that is suitable for the development of an event-driven application (e.g., GUI development) [18]. Using plain reactive programming (RP) makes it difficult to resolve data dependency due to the side-effects of each instruction. In contrast, no side-

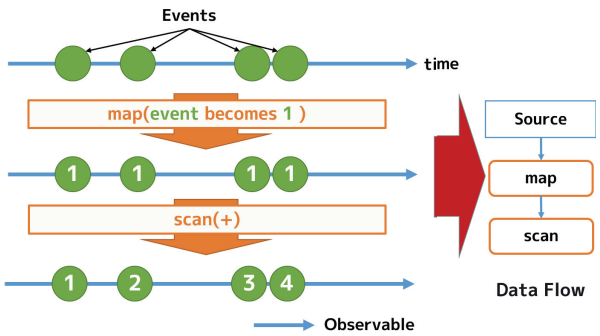

Fig. 7 An example of counting a number of events in the FRP.

```
1  input = MV::Observable.from_input
2  input
3    .map { |d| 1 }
4    .scan(0) { |reg, d| reg + d }
5    .subscribe { |d| print d }
```

Fig. 8 Sample code for implementing Fig. 7

effects exist with a "functional" RP. As a result, DFGs are more easily generated from a program written in FRP than one written in RP.

There is a data type that represents an "over time" value, referred to as an *observable*, such that programmers can declaratively describe operations for this data type. In imperative programming, interruptions are used by programs to realize dynamic variables. In contrast, when using FRP, programmers do not have to consider interruptions because the variables (objects) are updated over time.

We implemented our methodology in Ruby language [19] because Ruby offers flexible grammar and the capability of implementing DSL. Our proposal can be applied not only with Ruby but also with various programming languages, regardless of whether its type system is dynamic or static.

Let us investigate an example of an FRP. Figure 7 shows an example of an FRP while Fig. 8 shows its Mulvery program. In this example, the program counts a number of events. In line 1, an observable class is initiated as an object. Each instance of an observable has methods defined in the Rx for handling events that the observable arrived at by itself. Most Mulvery's operators takes lambda abstractions as arguments. Lines 3 and 4 of the sample code shows such a operators. A lambda abstraction is passed to the map operator in line 3 and, at any time, returns 1. A lambda abstraction is passed to the scan operator in line 4 and holds a result of an expression $reg+d$ in a register, i.e., "reg." In other words, the scan operator adds 1 to the variable *reg* each arrival of an event. As both the map and scan operator returns an observable object, operators can be chained. When a *terminal* event arrives at an observable, the observable is successfully completed. Thus, when the scan operator is complete, the subscribe method in line 5 calls a lambda abstraction that has been passed to itself. Finally, the lambda abstraction in line 5 prints the counting result.

Table 1 The principal schedulers in Mulvery.

Name	Roll
CurrentThreadScheduler	Pushes given process to the queue of the current thread
ImmediateScheduler	Starts given process immediately on the current thread
DefaultScheduler	Starts given process on a background thread
LocalScheduler	Creates a new thread and starts given process on it

The essential point is that the FRP concept is similar to a dataflow model. The FRP paradigm can express both temporal and spatial parallelism. Therefore, it is reasonable to assume that the FRP is also suitable to describe the behavior of hardware.

Lambda abstractions play a vital role in separating dataflow descriptions and data types. Each operator is abstracted to the operation level while the concrete manipulation of the data entities is handled by the lambda abstractions. Therefore, queries are able to fit various data formats with a uniform description method. Let us consider an *add* function as an example. The function is called without a lambda abstraction, i.e., as `obs.add(1)`, where the data types that the *add* function can handle are limited exclusively to numeric types. In contrast, if the function takes a lambda abstraction as an argument, the *add* function is available to handle any data type, even if it is a structure type as in the following code: `obs.add(1) { |d| d.value }`.

5. RTL Design Generation

This study focuses on the similarity between the development of programs using FRP and FPGA circuits and investigates the effectiveness of the FRP in synthesizing circuits from a program in dynamic typing languages, such as Ruby.

The steps shown next is the way to generate a circuit using Mulvery:

1. A user specifies an application's behavior with Mulvery operators,
2. Mulvery evaluates and analyzes the program written in step 1 on the Ruby interpreter to generate RTL codes, and
3. The user synthesize the RTL codes generated in step 2 with vendor tools.

5.1 Scheduling Strategy

To explain Mulvery's scheduling method, this subsection describes the concept of the Rx scheduler. A scheduler is a design-pattern of a class that schedules tasks to be processed. The most Rx operators require a scheduler object as an argument. This scheduler concept realizes multi-thread programming and provides priority to the tasks.

There are several schedulers that are implementations of frequently used strategies (e.g. "execute on a new thread" and "interrupt") in the pre-defined schedulers of the Rx. Table 1 shows the important schedulers and Fig. 9 shows the

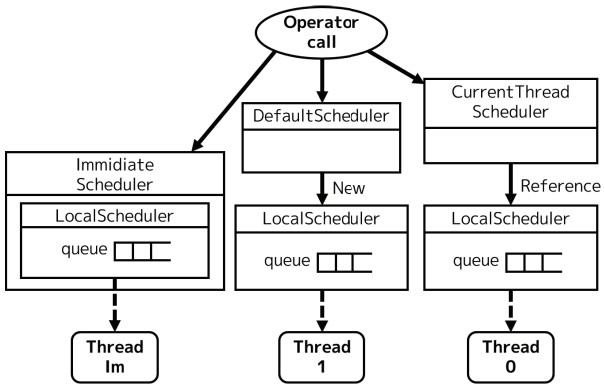

Fig. 9 The relationship among the operators, threads, and principal schedulers.

Table 2 Rx operator classification.

Creation-Class	Timer-Class	Data-Manipulation-Class
from_array	timer	select
from_IO	interval	map
empty		average

relationships among the operators, threads, and schedulers. Table 1 shows the important schedulers and Fig. 9 shows the relationships among the operators, threads, and schedulers. The *LocalScheduler* has a queue and executes tasks from the head of the queue. Each instance of the *LocalScheduler* works as a different thread.

Normally, on regular Rx, most operators use the *CurrentThreadScheduler* as the default scheduler. The *CurrentThreadScheduler* adds new tasks to the queue of the current thread. Therefore, a system sequentially calculates the given operators. We classified the Rx operators into three classes, as shown in Table 2, and reconsidered the default scheduler for each operator to realize automatic spatial parallelization. The operators classified to the Creation-Class use *DefaultThreadScheduler* as a default scheduler instead of the *CurrentThreadScheduler*. As the operators in the Creation-Class are data sources and have no dependency with the other operators in the Creation-Class, each operator in Creation-Class can be recognized as an independent dataflow.

Operators in the Data-Manipulation-Class remain to use the *CurrentThreadScheduler* as the default scheduler because these operators are sequentially executed from a data source.

Operators in the Timer-Class use the *DefaultThreadScheduler* as the default scheduler. These operators create a new thread because the timers are typically run on an independent thread.

5.2 Building a DFG

Mulvery executes the *build_dataflow* method of the given program to generate the HDL code. Thus, a chain of Mulvery's operators that specify a dataflow is dynamically evaluated at once when generating hardware. In our method, instances of the *DefaultThreadScheduler*, which are initiated in the Creation-Class operators, construct a dataflow from an Mulvery program. Therefore, every Rx query is recognized as an independent dataflow. This subsection explains how DFGs are generated with schedulers, as well as how an RTL design is generated from these DFGs.

```
1  module Observable
2    class << self
3      def from_array(array, \\
4          scheduler=LocalScheduler.new)
5        operator = \\
6            DataflowOperator.new(:from_array)
7
8        scheduler.schedule(operator)
9      }
10     ...
```

Fig. 10 An implementation of the *from_array* operator in Mulvery.

(1) Constructing a DFG

Dataflow graphs are composed of instances of the *Observable* class and *DataflowOperator* class. Figure 10 shows a part of the implementation of the *Observable* class. The *from_array* operator registers an instance of the *DataflowOperator* class instead of a task into its scheduler. Lambda abstractions given to an operator are recorded in an instance of the *DataflowOperator* (Fig. 13).

When the *DataflowOperator* is initiated, the instance registers itself in the *DataflowContainer*. Therefore, every independent dataflow is contained in the *DataflowContainer* class.

(2) Generating an RTL design from the DFG

Mulvery generates an RTL design written in Verilog from a dataflow graph saved in the *DataflowContainer*. As HDL code generation is separate from DFG generation, HDL code generation can be readily exchanged from Verilog HDL to VHDL or another HDL.

A template for HDL description is prepared for each Mulvery's operator. Each *DataflowOperator* (see Fig. 13) passes parameters to Ruby's template engine (ERB) to generate a module in Verilog using a corresponding template. At this time, the lambda abstractions saved in an instance of the *DataflowOperator* are translated into Verilog code, where the generated code passes to the ERB. The ERB embeds the code in the template corresponding to the instance of the *DataflowOperator*, as shown in Fig. 11 and Fig. 12. At this point, to exchange data and instructions with one or several CPUs, Mulvery allocates an address on the CoRAM for each observable object if it is referenced from software part.

5.3 Circuit Synthesis from Lambda Abstractions

Mulvery's operators defined as high-order functions, such as a map or scan, take a lambda abstraction as an argument. A lambda abstraction that is passed to an operator takes an event as its argument. For example, Fig. 14 shows a program

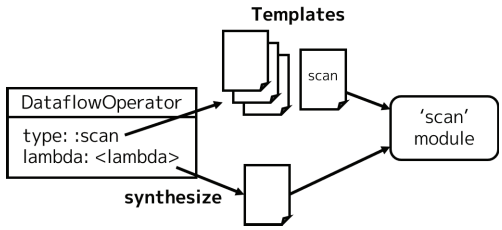

Fig. 11 The workflow that generates the HDL code of a module.

```
case operator.name
...
when :buffer then
  self.append_module_hdl( \\
      ERB.new(BUFFER_HDL).result(binding))
...
```

Fig. 12 Part of the code for the HDL generation of an operator.

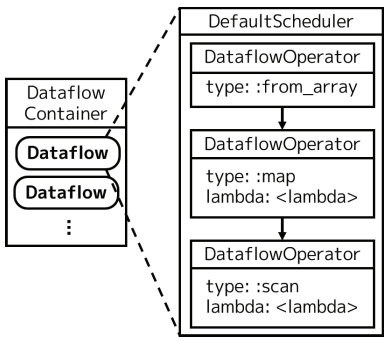

Fig. 13 The structure of a dataflow and its container.

```
sum = observable.map(){ |event|
  event * 2
}
```

Fig. 14 Sample code of the *map* operator.

that doubles data arriving as an event. Operators (functions) can receive a lambda function as a block in Ruby's syntax. Although such a lambda abstraction is not written with FRP, it also must be translated into HDL codes. Hence, we have employed another way to realize parallel computing for the lambda abstractions. Each data object in a program (i.e., the variable *event* in Fig. 14) has meant an edge of the dataflow in its DFG.

(1) Generating DFGs from lambda abstractions

For static, high-level synthesis of a lambda abstraction into a circuit, a synthesizer must infer the types of all variables. However, Ruby is not a static typing language. Thus, for the sake of simplicity, Mulvery regards the lambda abstraction as a dataflow that initiates from its arguments. The map operator is implemented as shown in Fig. 15. A piece of the *HDLComposer* implementation class is also shown in Fig. 12. The *yield* statements used in an operator call and evaluate a block given to the operator. To generate a piece

```
def map()
  lambda_rtl = yield(HDLComposer.new())
  self.append_module( \\
      ERB.new(MAP_HDL).result(binding))
end
```

Fig. 15 An implementation of the *map* operator.

```
s.map() begin |value|
  mv_if(value, ->v{v == 1}){
    value * 1
  }
  .elsif(->v{v == 2}){
    value * 2
  }
  .else{
    value * 3
  }
  .endcheck
end
```

Fig. 16 Sample code of the special "if" statement provided in Mulvery.

of HDL code from a lambda abstraction, Mulvery passes an *HDLComposer* object for each argument of the lambda abstractions. The *HDLComposer* behaves as a numeric-type object by default but users can implement other object behaviors if necessary.

(2) Control statements

The method described in the paragraph above generates unexpected HDL code when Ruby's control statements, such as *if* or *for*, are used in the lambda abstractions because execution is branched at these statements. Thus, Mulvery provides special methods and classes to use control statements. An example of an *if* statement in Mulvery is shown in Fig. 16. The *mv_if* method returns an *IfContext* object, resulting in the synthesis of other branches that are contained in the context object. The *IfContext* class has *elseif* and *else* methods. Thus, a programmer can use conditional statements.

6. Sample Application and Evaluation

To show our tool does not decrease the performance of an input program than the performance when the given program is executed as software, we developed an application with convolution operations for image processing to evaluate our proposed method. The application applies a Laplacian filter with a size of 5×5 to an image with a size of 128×128 (Fig. 17). Figure 18 shows the architecture of the application while Fig. 19 shows the essential code of the sample application, which receives a line (128 px) of an image as an event. When five events (i.e., five lines) are stored in the input buffer, the system simultaneously applies the Laplacian filter 124 times in parallel and emits a line of the processed picture as an event. Finally, the result buffer emits a processing result as an event when it stores 124 lines.

In this example, the *Matrix* class that is a standard class of Ruby is overridden by Mulvery library to be a hardware

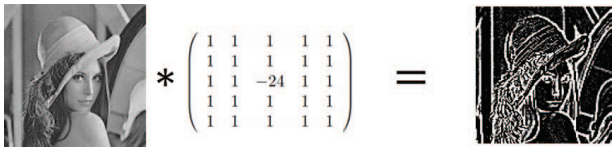

Fig. 17 The application of a 5 × 5 Laplacian filter.

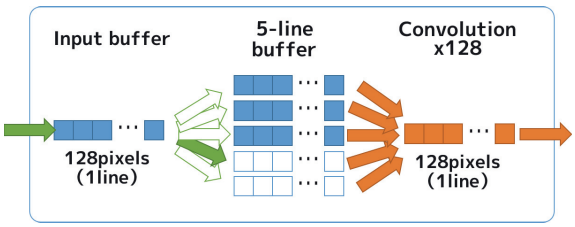

Fig. 18 The architecture of the sample application.

```
1  buffer = input.shift_buffer(5)
2  line = buffer.map() do |data|
3    mats = Array.new()
4    (0...(128-4)).each do |i|
5      mats.push(Matrix[data[0][i, 5], \
6                       data[1][i, 5], \
7                       data[2][i, 5], \
8                       data[3][i, 5], \
9                       data[4][i, 5]])
10   end
11   result_row = Array.new()
12   mats.map() do |mat|
13     result_row.push(mat.conv(kernel))
14   end
15 end
16 result = line.buffer(120)
```

Fig. 19 The essential part of the sample application.

module, and *conv* has been newly added to the *Matrix* class.

6.1 Structure of the Generated Hardware

Figure 20 and Fig. 21 shows the structure of the hardware generated by Mulvery. Mulvery generated an array consist of 5 × 128 register, denoted as *data*, and connected the array to a matrix consist of 5 × 5 pixels, denoted as *mats*. Each element of *mats* has been connected to each of 124 independent convolution arithmetic units. The convolution unit convolves an element of the *mat* with the matrix of the Laplacian filter saved as a variable *kernel*.

6.2 Scheduling Process

In the sample program shown in Fig. 19, the *from_array* operator reserves a new thread using *DefaultThreadScheduler*. After this, the map and the buffer operators chained to the *from_array* operator schedule itself to the thread reserved by *from_array* using *CurrentThreadScheduler*. A data object (e.g., the variable *data* and each element of the array *mats* in Fig. 19) will be an edge in the DFG. In other words, lines 4–10 in Fig. 19 generates 124 threads and the lambda abstraction defined in lines 12–14 runs on each thread. Therefore, the convolution mat.conv(kernel) runs on 124 threads in parallel.

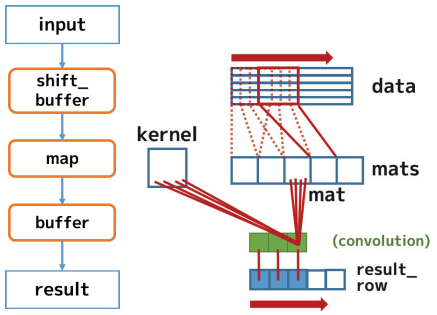

Fig. 20 The pipeline (left) and result of lambda abstraction synthesis (right) in Fig. 19.

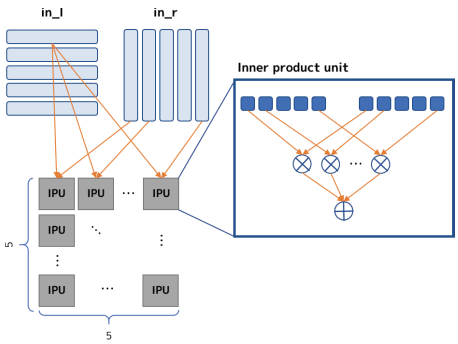

Fig. 21 Structure of matrix convolution unit (left) and structure of inner product unit (IPU) (right).

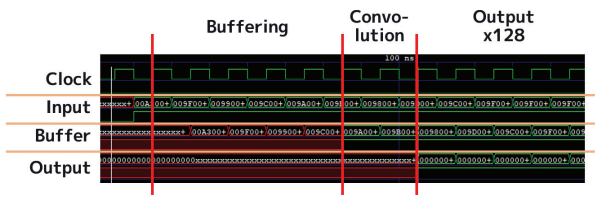

Fig. 22 A confirmation of hardware operation synthesized using the Icarus Verilog.

6.3 Evaluation

We evaluated the hardware generated from the sample application.

(1) Behavior verification

We verified the behavior of the hardware with the Icarus Verilog, whose results are shown in Fig. 22. One-hundred and twenty-eight iterations of the 5 × 5 convolution were completed in one clock. Mulvery generated high-throughput hardware without exploring the architecture. The first convolution required two clocks as only the first signal was skipped due to the influence of a rising clock.

(2) Resource consumption

Table 3 shows the resource consumption on the SoC FPGA

Table 3 The resource consumption of the Zynq-7000 (XC7Z020).

Resource	Estimation	Available	Utilization %
LUT	48,839	53,200	91.80
FF	3,949	106,400	3.71
BRAM	35.50	140	25.36

Zynq-7000 (XC7Z020) platform released by Xilinx, Inc. Because many of matrix multipliers had been generated, the LUT consumption was huge. Mulvery has no way to directly control resource consumption and calculation speed. The resource consumption control is one of the current issues of Mulvery.

(3) Software comparison

Finally, we compared the processing speed with the case where the software was operated on a CPU. Ruby was implemented on an Intel Core-i3 at 3.4 GHz, where the *perf* command on CentOS 7 was used to measure the runtime.

The result was that the sample application required 3,130 ms of CPU time to process the image. In contrast, the generated hardware consumed approximately 135 clocks per image (Fig. 22). As the operating frequency of the FPGA was 100 MHz, the maximum throughput was approximately 194 GB/s. However, it is difficult to input images at 194 GB/s from the memory or from an external I/O to the FPGA. On the Zynq platform, the measured bandwidth for reading memory was approximately 1.6 GB/s. Thus, the data buses were bottlenecked, such that the actual maximum processing speed was approximately 0.2 ms per image as the data transferred from the memory.

Despite problems with bottlenecked data buses, Mulvery can generate hardware compared with standard CPU processing, without explicit circuit synthesis tuning by a designer.

7. Conclusions

This paper proposed a hardware and software co-design environment, named *Mulvery*. FRP, which is widely used for software development, was applied to our tool since the semantics of FRP programs is similar to hardware design.

According to the experiment, the method allows us to design hardware without degradation of performance. A sample application that applied a Laplacian filter to a 128×128 image processed the convolution operation within one clock, with a 91.8% resource consumption on the Zynq-7000 (XC7Z020) platform.

We introduced a method to generate DFGs from a program based on FRP. To our knowledge, no other hardware design framework based on sofware development techniques, i.e., FRP and a dynamic typed language, exists. Mulvery generates a circuit for an FPGA from query chains written in the pre-defined operator set in Mulvery. Mulvery's pre-defined query set is based on ReactiveX, such that Mulvery translates each query defined in the query set to a hardware module. Mulvery converts a Ruby program into DFGs and then converts the DFGs into Verilog code. Software-like development which adopts a dynamic typing language suffers from a lack of previous studies. Our method, based on the FRP, can be applied to numerous programming languages, regardless of whether the system is dynamic or static. Therefore, our proposal can be used not only with Ruby but also with various programming languages.

Acknowledgments

This work was supported by the JSPS KAKENHI, Grant Numbers 16K00078 and 18J22381. Additionally, certain components of this study are the results of research related to the "IPA MITOU Project" and the "JSPS Overseas Challenge Program for Young Researchers."

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for fpga-based processor/accelerator systems," Proc. 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11, New York, NY, USA, pp.33–36, ACM, 2011.
[2] Maxeler Technology, "MaxCompiler White Paper," http://www.maxeler. com/, 2011.
[3] Intel, "Intel FPGA SDK for OpenCL," https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, Ref. Aug. 2019.
[4] Xilinx, "Vivado HLS," http://xilinx.com/products/design-tools/vivado/integrationesl-design.html, Ref. Aug. 2019.
[5] S.W. Ambler and M. Lines, eds., Disciplined Agile Delivery — A Practitioner's Guide to Agile Software Delivery in the Enterprise, IBM Press, 2012.
[6] MIT Artificial Intelligence Lab, Dynamic Languages Group, "Call for participation of 'll1: Lightweight languages workshop'," http://ll1.ai.mit.edu/, Ref. Aug. 2019.
[7] "ReactiveX," http://reactivex.io/, Ref. Aug. 2019.
[8] C. Elliott and P. Hudak, "Functional reactive animation," Proc. second ACM SIGPLAN international conference on Functional programming - ICFP '97, pp.263–273, 1997.
[9] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," Proc. ACM SIGPLAN workshop on Haskell - Haskell '02, pp.51–64, 2002.
[10] J. Liberty and P. Betts, eds., Programming Reactibe Extensions and LINQ, Apress, 2011.
[11] E.S. Chung, J.C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," Proc. 19th ACM/SIGDA international Symposium on Field Programmable Gate Arrays - FPGA '11, New York, NY, USA, pp.97–106, ACM, 2011.
[12] D. Lockhart, G. Zibrat, and C. Batten, "PyMTL: A unified framework for vertically integrated computer architecture research," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp.280–292, Dec. 2014.
[13] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," Proc. 49th Annual Design Automation Conference on - DAC '12, pp.1216–1225, June 2012.
[14] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D.A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, April 2016.

[15] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog HDL," Applied Reconfigurable Computing, Lecture Notes in Computer Science, vol.9040, pp.451–460, Springer International Publishing, Cham, April 2015.
[16] Xilinx, "SDAccel," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, Ref. Aug. 2019.
[17] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," Computer, vol.41, no.7, pp.40–46, July 2008.
[18] E. Bainomugisha, A.L. Carreton, T.V. Cutsem, S. Mostinckx, and W.d. Meuter, "A survey on reactive programming," ACM Comput. Surv., vol.45, no.4, pp.1–34, Aug. 2013.
[19] Ruby community, "Ruby Programming language," https://www.ruby-lang.org/en/, Ref. Aug. 2019.

Daichi Teruya is currently a Ph.D. student at the Tokyo University of Agriculture and Technology, Graduate School of Engineering. In 2015, he received a fundamental degree of engineering from the National Institute of Technology, Okinawa College. In 2017, he received a bachelor degree of engineering from Tokyo University of Agriculture and Technology, Department of Information Engineering. In 2018, he received a master's degree of engineering from Tokyo University of Agriculture and Technology.

Hironori Nakajo received B.E. and M.E. degrees in Electrical Engineering from Kobe University in 1985 and 1987, respectively. He was a Visiting Research Assistant Professor at the Center for Supercomputing Research and Development (CSRD) at the University of Illinois at Urbana-Champaign from 1998 to 1999. Since 1999, he has been an Associate Professor at the Institute of Engineering, Graduate School, Tokyo University of Agriculture and Technology. His research interests include computer architecture, parallel processing, cluster computing, and reconfigurable computing. He is a member of the IPSJ, IEEE, and ACM. Ph.D. (Doctor of Engineering).