A Haskell-Based Programming Paradigm for Coarse-Grained Reconfigurable Arrays

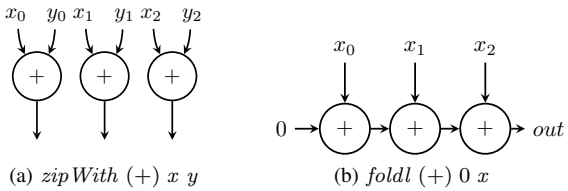
Anja Niedermeier, Jan Kuper, Gerard Smit
University of Twente, Department of Computer Science
Enschede, The Netherlands
a (dot) niedermeier (at) utwente (dot) nl

I. Motivation

Programming coarse-grain reconfigurable arrays (CGRAs) is a challenging task [1] , [2]. In this work, we exploit the algebraic structure which is often present in the specification of a streaming application to distribute the different parts of a computation over a multi-core architecture. This architecture is dataflow-based, so that the control of the cores coincides with the availability of data on the channels. In this paper, we focus on the compiler, not the architecture. We formulate our work in the functional programming language Haskell since that is close to a mathematical formalism.

II. Haskell

A powerful feature of a functional language is the availability of higher-order functions, which not only accept values, but also functions as arguments. Figure 1 shows two examples of such higher order functions: *zipWith* takes two vectors x and y as arguments and applies a specific operation (+) pairwise to the elements in the vectors, whereas *foldl* accumulates the values in a vector x by means of a specific operation (+), starting from an initial value (0). Note that the arguments to these functions are separated by spaces, and not by commas.



(a) $zipWith\ (+)\ x\ y$ (b) $foldl\ (+)\ 0\ x$

Figure 1: Higher order functions

Another powerful feature is the possibility to define embedded domain specific languages (EDSL) as data-types within Haskell. First of all, a parser for such an EDSL comes for free, and transformations of expressions in an EDSL can be defined as normal functions which exploit the practical feature of pattern matching.

III. Target architecture

We have been developing an architecture for efficient execution of streaming applications containing a high degree of instruction-level parallelism. This paper is about the compiler for this architecture, therefore the architecture will only be outlined in brief terms.

The architecture belongs to the class of coarse-grain reconfigurable arrays (CGRAs) and is composed of a mesh of simple, configurable cores. The size of the mesh is configurable at design time, for this paper we are using a mesh of 16 cores. The cores include a functional unit for binary operations, a local register file and a program memory which is used to control the behaviour of the core. The cores are interconnected by both a nearest-neighbour network as well as a global network on chip (NoC) for full connectivity. The programming of each core is based on finite state machine (FSM) logic where each state is again defined by means of a specific instruction set. The execution mechanism of each core is based on dataflow principles, i.e. as soon as the required input data is available, the nodes executes.

IV. Compiler Design

In this section, the proposed compiler is outlined and the implementation is briefly explained. First, the actual language is shown, which is implemented as Embedded Domain Specific Language (EDSL). Then, we show how this EDSL is used to implement functions and built an abstract syntax tree (AST). Finally, we outline how the AST is mapped to the target architecture.

We first define an elementary version of the programming language for our target architecture as an EDSL in Haskell, by means of the algebraic data type *Expr* (see Listing 1), containing a constructor for every operation a core in the target architecture can execute. In the language *Expr* constants can be formulated (line 1), delays and feedback loops are supported (lines 2 and 3) and inputs can be represented (line 4). Note that the EDSL is a *recursive* data type, i.e., the type *Expr* is used in the defintion of *Expr* itself.

```
data Expr = Const Int                          1
          | DELAYED Expr                       2
          | FEEDBACK                           3
          | Input String                       4
          | Op OpCode Expr Expr                5
data OpCode = ADD | MUL | SQR | AND ...        6
```

Listing 1: recursive EDSL definition for an expression

An example of the usage of the language *Expr* to define a sum of a constant and an input is as follows:
$sum = Op\ ADD\ (Const\ 1)\ (Input\text{“}x'')$

Regular structures can be expressed by using Haskell's higher order functions, as in the accumulation of addition over a sequence x of numbers:

$sum\_up\ x = foldl\ (Op\ ADD)\ (Const\ 0)\ x$

A further advantage of using an algebraic data type for the EDSL is that an expression formulated in the language in fact already *is* an AST. That is to say, we don't need to define a parser. For example, the AST of the above function sum_up is automatically generated when evaluating sum_up:

```
ghci> sum_up [Input "x0",Input "x1",Input "x2",Input "x3"]
ghci> Op ADD
            (Op ADD
                  (Op ADD
                        (Op ADD (Const 0) (Input "x0"))
                        (Input "x1"))
                  (Input "x2"))
            (Input "x3")
```

As final step, the AST is implemented on the architecture. The implementation consists of two steps: First each node in the AST is mapped to one core in the architecture, then the code for each core and the routing information are generated. The mapping is currently performed with an ILP formulation, but since the focus of this paper is on the compiler, it is not outlined further here. To generate code for the cores, the AST is traversed from the root node and for each node the corresponding code is generated according to the programming principle mentioned in Section III.

V. Usage

To illustrate the usage principle of the complete design flow, we show how a multiplication of two vectors can be specified in the presented EDSL and then mapped to the target architecture.

The multiplication of two vectors vxv, shown in Listing 2, is built using the two higher-order functions $zipWith$ and $foldl$ that have already been explained in section II. A graphical representation of vxv for two vectors of length eight is shown in Figure 2.

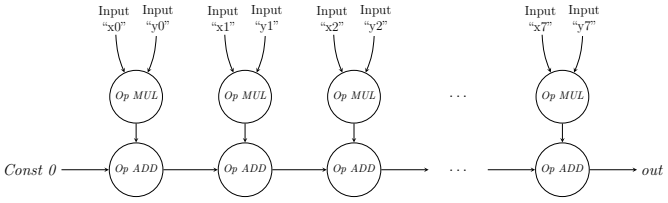


Figure 2: Structure of vxv

```
vxv x y = out                                  1
  where                                        2
    ms = zipWith (Op MUL) x y                  3
    out = foldl (Op ADD) (Const 0) ms          4
```

Listing 2: Implementation of a multiplication of two vectors

In line 1 of the code, the function name vxv and its arguments x and y which are the two vectors to be multiplied are defined. out is the resulting output. In line 3, the vectors are element wise multiplied which leads to the row of $Op\ MUL$ in Figure 2. Finally, in line 4, the results of the multiplications are added up, which leads to the row of $Op\ ADD$ in Figure 2. Note that the input vectors are now no concrete numbers as in the example in Section II, but instead represent input signals denoted with *Input "name index"*.

The next step in the design flow is the actual code generation and mapping to the target architecture. The mapping for vxv is shown in Figure 3.

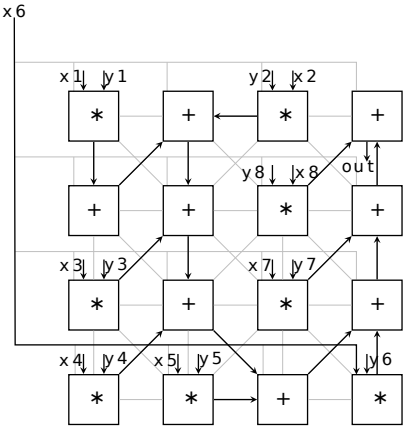


Figure 3: Mapping of vxv

For testing purposes, a simulation function is provided by the compiler where the implemented function can be tested with concrete input values. To test the function vxv with the two test vectors $x = [1, 2, 3]$ and $y = [4, 5, 6]$, the simulate function $simExpr$ is called which then displays the result:

```
ghci> simExpr vxv x y
ghci> 32
```

Furthermore, the compiler provides auto generated graphical representations of both the AST and the mapping result.

VI. Conclusion

A compiler was developed for an existing coarse-grained reconfigurable architecture to implement functions with a high degree of instruction-level parallelism. For this, the functional programming language Haskell was used, as it inherently has a notion of structure and, thus, can easily express parallelism and the flow of data. For the compiler, we implemented an embedded domain specific language (EDSL) as recursive datatype. In combination with higher order functions, this EDSL can be used to construct expressions that directly resemble the structure of a given problem and thus can be mapped in a straight-forward way to the target architecture.

References

[1] C. Brunelli, F. Garzia, J. Nurmi, F. Campi, and D. Picard, "Reconfigurable hardware: The holy grail of matching performance with programming productivity," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 409–414.

[2] B. Svensson *et al.*, "Evolution in architectures and programming methodologies of coarse-grained reconfigurable computing," *Microprocessors and microsystems*, vol. 33, no. 3, pp. 161–178, 2009.