
Author #1

Abstract:
Dataflow Designer is a new rapid prototyping tool and graphical Integrated Development Environment (IDE) for the programming of coarse-grained architectures. Its main and novel advantage is the use of rewriting logic strategies for the automated generation of functionally equivalent alternative implementations from a single mathematical specification. These alternative implementations are then compiled using a retargetable dataflow compiler, which generates the final code for the targeted architecture. The design flow requires only minimal user intervention, mainly the selection of the rewriting logic strategies to be applied and the evaluation of results from the final simulation. Dataflow Designer can also be used for coarse-grained design space exploration through its interface to the KressArray Xplorer. The current version of the tool is targeted for the eXtreme Processing Platform from Pact, with support for additional architectures planned in future versions. A FIR Digital Filter is presented as a proof of concept to illustrate some of the capabilities of the tool and compare it with the available commercial tools for the XPP.

1 Introduction:

There is growing interest in coarse-grained reconfigurable arrays because they offer increased power and better area efficiency than fine-grained reconfigurable arrays [1]. A coarse-grained array is defined as an array of functional elements with pathwidths greater than 1 bit, in other words, reconfiguration is done at the function element level instead of the gate level.

Working at the functional level allows an easier application of approaches taken from mathematics and theoretical computer science, like formal methods and rewriting-logic. Rewriting has very simple operational semantics, which are based in matching and simplification (reduction), offers represents a natural mechanism for manipulation of algebraic expressions and is considered the formal framework for reasoning about the functional programming paradigm [2, 3]. Because of its simplicity, describing computational (hardware-software) behavior using rewriting avoids the inclusion of the unnecessary operational semantics necessary in programming languages. Using rewriting in the Dataflow Designer allows the specification of mathematical functions in a clean manner and the generation of alternative ways of computing it, which result from the application of this simple operational semantics.

When prototyping a given mathematical function, it is necessary to check different equivalent alternatives of computing it. In the mathematical context, rewriting allows for reduction of algebraic expressions until reaching the "simplest" equivalent forms. This simplicity is measured according to the intuition that when applying a reduction step to a term this is being transformed into a simpler one. By combining rewriting with logic strategies, rewriting(-logic) allows for different paths of simplification. This is done by selecting different strategies for controlling the application of the rewriting rules. This enables the generation of different alternative equivalent "simplified" versions of the original mathematical expression, which are by definition “correct-by-design” and can be further synthesized and tested using other tools.

By enlarging rewriting with logic, which is called rewriting-logic, it is obtained a natural mechanism of discriminatingly representing the behavioral changes introduced by reconfiguration over flexible architectures. The foundational mathematics of rewriting theory combined with the versatility of logic (plus the type manipulation included in rewriting-logic computational environments) have been proved of great usefulness in the prototyping of dynamically reconfigurable architectures developed for space efficient computation of non trivial algebraic operator such as the FFT [4] and for implementing run-time efficient dynamical programming methods for sequence comparison [5].

1.1 Rewriting logic

In the architectural context, rewriting rules are of the form \( s \Rightarrow t \mid C \), where \( s \) and \( t \) are terms over a given signature, the left- and right-hand side of the rule, and \( C \) is a premise. Usually, these rules are syntactically restricted in such a way that variables occurring in the right-hand side of the rule and in the premise occur in the left-hand side too. A given term can be reduced using such a rewriting rule, whenever the left-hand side of the rule matches a sub-term of the given term and the corresponding instance of the premise holds. Thus, the given term is reduced by substituting the matched sub-term by the corresponding instance of the right-hand side of the rule. These operational semantics are the same involved in functional
environments and has been promoted in functional programming languages since the well-known McCarthy LISP of the 1950s [6]. In the algebraic context, pure rewriting is useful for reaching canonical or normal forms of a given term; that is, reaching the "simplest" representation corresponding to a given algebraic expression. But in the architectural context, it is well-known that the "simplest" expression does not necessarily coincide with the more adequate form to implement the expression. Thus, combining rewriting with logic strategies result increases the capabilities for determining different alternatives of representing these operators; the logic strategies control the application of the rewriting rules, allowing the generation of different canonical forms for the same mathematical expression. In this way, we obtain a formal mechanism for producing different canonized versions of a mathematical operator, which can be quickly prototyped using other design tools. The most popular rewriting-logic computational environments are Maude [7], ELAN [8] and CafeOBJ [9]. The first has been applied in the design and formal verification of processors (via model checking implemented by rewriting) [10], while the second in the modeling and simulation of reconfigurable hardware [4,5].

1.2 Pact eXtreme Processing Platform (XPP)

Pact XPP is an array of coarse-grain, data-driven, runtime reconfigurable ALU and RAM elements [11]. A simplified structure of an XPP array is shown in Figure 1 [12]. The XPP offers very powerful and fast mechanisms for dynamic reconfiguration, which allows the use of relatively small arrays to obtain a very high performance per energy ratio.

![Figure 1. Simplified Structure of the XPP Array](image)

The ALUs can perform arithmetic and logical functions, including addition, subtraction, multiplication, shifting, AND, OR, complex addition and multiplication, swap, and some other specialized functions. Each ALU Processing Array Element has also a Forward Register (FREG) and Backward Register (BREG) that are used for routing. Additionally the BREG can also perform addition and subtraction.

The XPP is available as a synthesizable IP core with arbitrary sizes and data-widths. Pact also sells a XPP device called the XPP64A, which offers an 8x8 array of ALU elements, 16 RAM elements and 4 I/O elements, all of them with a 24 bit data-width. The XPP is programmed using Pact’s Native Mapping Language (NML), while a Vectorizing C-Compiler to NML is also available [12].

2 Related work:

Diverse tools and approaches have been used to generate code for coarse-grained architectures. A very interesting approach is the retargetable dataflow compiler, which uses a description of the target architecture in a separate hardware file in order to allow the use of the same program for different coarse grained architectures. Two relevant examples of such retargetable compilers include the KressArray Xplorer [13,14] and IMEC’s DRESC Compiler [15]. The Pact XPP VCC Compiler [12] uses automated loop unrolling and loop vectorization to parallelize a C program and map it into the XPP. Another interesting approach is taken by COMPAA/LAURA[16] toolset, which extracts the dataflow and control code from a mathematical model created in Matlab/Simulink and converts it directly into synthesizable VHDL code.

Recent work on rewriting based treatment of hardware design includes the work from Kapur, who used the well-known Rewriting Rule Laboratory - RRL for verifying arithmetic circuits [17]. Arvind applied rewriting in the specification of processors with simple architectures, the rewrite-based description and synthesis of simple logical circuits and the description of cache protocols over memory systems [18]. In his work, terms and rewriting rules were used to describe hardware states and behavior. Rewriting-logic has been shown to have greater flexibility than pure rewriting for the discrimination between fixed and reconfigurable elements of reconfigurable architectures, allowing for a natural and quick conception and simulation of implementations of new reconfigurable computing paradigms. Applications of rewriting-logic in this architectural design context include [Me03] and [ARNa03,ARJa04] mentioned previously.

3 Dataflow Designer

Dataflow Designer is a new rapid prototyping tool and graphical Integrated Development Environment (IDE) for the programming of coarse-grained architectures. Its main and novel advantage is the application of rewriting logic strategies for the automated generation of functionally equivalent alternative implementations from a single mathematical specification. These alternative implementations are then compiled using a retargetable dataflow compiler, which generates the final code for the targeted coarse-grained architecture. All the steps of the design flow are fully automated and integrated, requiring only minimal user intervention for the selection of the rewriting logic strategies to be applied and the evaluation of results from the final simulation.

Dataflow Designer receives a mathematical description and generates alternative implementations based on the rewriting-logic strategies selected by the user. The rewriting strategies are based on rewriting rules, which
The automated mathematical manipulation/optimization is done through rewriting-logic, using pre-defined rules and strategies. Applying different strategies result in alternative implementations that are functionally equivalent to the original specification. Strategies may contain any combination of symbolic mathematical manipulation and the substitution for different implementations of the same sub-functions or operators. Different data types and their operators can be specified independently. A data-type can also be specified as a subset or superset of other data-types.

3.1 Implementation

All steps of the current design flow are shown in figure 2, and explained in the following subsections.

**Mathematical Equation**

![Diagram](image)

**Figure 2. Design Flow Block Diagram**

3.1.1 Mathematical manipulation/optimization

The automated mathematical manipulation/optimization is done through rewriting-logic, using pre-defined rules and strategies. Applying different strategies result in alternative implementations that are functionally equivalent to the original specification. Strategies may contain any combination of symbolic mathematical manipulation and the substitution for different implementations of the same sub-functions or operators. Different data types and their operators can be specified independently. A data-type can also be specified as a subset or superset of other data-types.

Figure 3 shows the definition of some of the operators of the XPP and their effects on the data types: “constant” and “variable”, while all inputs and outputs are defined to be variables. Operators can be defined to be also associative (to the left or to the right) and/or commutative. Conflicts between operators are controlled by giving them priority values to indicate their hierarchy.

```plaintext
rules for variable
a : constant;
x, y : variable;
global
[expandrules] z(a*x) => a*z(x) end
   [expandrules] z(a+x) => a+z(x) end
   [expandrules] z(y*x) => z(y)*z(x) end

   [collapse] a*z(x) => z[a*x] end
   [collapse] z(y)*z(x) => z(y+x) end
   [collapse] a*z(x) => z[a*x] end

end

strategies for variable
implicit
   [] expand => normalise(expandrules) end
   [] collapse => normalise(collapse) end

end
```

Figure 4 shows the definition of some basic rewriting rules and strategies, which will be demonstrated on a FIR filter to be explained in the next section.

3.1.2 ALEX code generator

This stage will automatically convert the previous stage result into compilable ALEX code for the KressArray Xplorer. The ALEX code generator first writes all input, outputs and constants as defined in the original mathematical description specification file, and then will create temporal variables and parse the mathematical equation to eliminate any recursivity.
3.1.3 Retargetable dataflow compiler

The ALEX retargetable compiler from the KressArray Xplorer is used for this step. A retargetable dataflow compiler allows the generation of code for different target architectures by changing the target hardware description in the hardware file.

Additionally, the interfacing with the KressArray Xplorer allows the use of Dataflow Designer for design space exploration of coarse-grained architectures.

The ALEX compiler requires two inputs: the Algorithm (written in ALEX language) and the Target Architecture Hardware File. The output of the compiler will be a dataflow representation (called alphaview) of the algorithm using the operators and functions defined in the hardware file. The ALEX language is based on a subset of the C language. The Hardware File for the XPP architecture consists of the definition of the arithmetical and logical operators of the ALU blocks and external submodules. External submodules can be used in the compilation by defining them as functions in the hardware file.

The mapping and simulation is done using the Pact Software Development Suite PSDS-M64 for the XPP.

4 Proof of concept results

The above methodology was tested using a Four Tap Finite-Impulse-Response (FIR) Filter as a proof of concept. In this case, the goal was to automatically obtain a timing-optimized implementation using only the direct form mathematical equation of the FIR Filter.

4.1 Input mathematical equation:

The direct-form of a four-tap FIR Filter [19] is defined by:

\[ y_i = \sum_{k=0}^{d} b_k x_{i-k} \]

For implementation purposes, we define:

\[ z(x_1) = x_{1-1} \]

Then, the direct form equation can be written as:

\[ y_i = b_0 x_i + b_1 z(x_1) + b_2 z(x_2) + b_3 z(x_3) + b_4 z(x_4) \]

This will be used as the input for the rewriting-logic step of the design-flow. A graphical representation of this equation can be seen in Figure 6.

4.2 Manipulation/optimization by ELAN:

The “collapse” strategy is selected from the strategies list of dataflow designer. An extract of the resulting output can be seen in Figure 7.

The resulting dataflow file consists in a list of linked operator nodes and functions. Each of these operator nodes corresponds to an XPP ALU or BREG, while a function node corresponds to an external NML module.

3.1.4 NML code generation from the dataflow:

Since the dataflow compiler is already mapping to the XPP operators specified in the Hardware File, the resulting dataflow can be mechanically converted into the NML language, and the generated NML modules can be directly mapped on the XPP.

3.1.5 Mapping and simulation

The mapping and simulation is done using the Pact Software Development Suite PSDS-M64 for the XPP.

4 Proof of concept results

The above methodology was tested using a Four Tap Finite-Impulse-Response (FIR) Filter as a proof of concept. In this case, the goal was to automatically obtain a timing-optimized implementation using only the direct form mathematical equation of the FIR Filter.

4.1 Input mathematical equation:

The direct-form of a four-tap FIR Filter [19] is defined by:

\[ y_i = \sum_{k=0}^{d} b_k x_{i-k} \]

For implementation purposes, we define:

\[ z(x_1) = x_{1-1} \]

Then, the direct form equation can be written as:

\[ y_i = b_0 x_i + b_1 z(x_1) + b_2 z(x_2) + b_3 z(x_3) + b_4 z(x_4) \]

This will be used as the input for the rewriting-logic step of the design-flow. A graphical representation of this equation can be seen in Figure 6.

4.2 Manipulation/optimization by ELAN:

The “collapse” strategy is selected from the strategies list of dataflow designer. An extract of the resulting output can be seen in Figure 7.

The resulting equation:

\[ y = (b_0 x_i) + z((x*b_1) + z((x*b_2) + z((x*b_3) + z((x*b_4))))) \]

Which corresponds to a timing optimized version of the FIR Filter. A graphical representation of this equation can be seen in Figure 8.

Applying the “Expand” rewriting strategy to this equation would result in the original direct form.
4.3 Conversion to ALEX code:

Figure 9 shows the resulting ALEX code. The inputs and outputs are defined in the respective ScanWindow sections. Temporal variables were added by the converter to eliminate recursive function calls.

```c
rALUSubnet SubNet_0 (In,Out)
  ScanWindow In
  { int x at [0][0];
    HandleOffset [0][1];
  }
  ScanWindow Out
  { int y at [0][0];
    HandleOffset [0][1];
  }
  const int b0=100;
  const int b2=102;
  const int b3=103;
  const int b4=104;
  int aux0,aux3,aux4,aux6,aux7,aux9,aux10,aux12;
  aux0=x*b4;
  aux3=x*b3;
  aux4=(aux3)+z2(aux0);
  aux5=x*b2;
  aux7=(aux6)+z2(aux4);
  aux9=x*b1;
  aux10=(aux9)+z2(aux7);
  aux12=b0*x;
  y=(aux12)+z2(aux10);}
```

Figure 9. Generated ALEX code

4.4 Compilation of ALEX code:

Figure 10 shows the alphaview code of the resulting dataflow using the XPP operators. The ALUs and their operations are defined with OPERA, the used submodules with FCALL, the constants with CONST, the inputs with VAREF and the outputs with VDEFN.

```c
...CONST $001 100 0;
CONST $002 101 1;
CONST $003 102 2;
CONST $004 103 3;
CONST $005 104 4;
VAREF $006 'W (G) [0][0] 1n'0 0 0;
OPER A $007 'MUL' ' ' ($001 $006) 8;
OPER A $008 'MUL' ' ' ($006 $002) 8;
OPER A $009 'MUL' ' ' ($006 $003) 8;
OPER A $010 'MUL' ' ' ($006 $004) 8;
OPER A $011 'MUL' ' ' ($006 $005) 8;
FCALL $012 'Z'
  ($011) 16;
OPER A $013 'BREG_ADD' ' ' ($010 $012) 4;
FCALL $014 'Z'
  ($013) 16;
OPER A $015 'BREG_ADD' ' ' ($009 $014) 4;
FCALL $016 'Z'
  ($015) 16;
OPER A $017 'BREG_ADD' ' ' ($008 $016) 4;
FCALL $018 'Z'
  ($017) 16;
OPER A $019 'BREG_ADD' ' ' ($007 $018) 4;
VDEFN $020 $019 'W (G) [0][0] 1out'0 0 0;...
```

Figure 10. Resulting Dataflow (Alphaview) code

4.5 Conversion to NML and mapping:

The resulting dataflow of the previous step can be automatically translated into NML by the conversion tool, and later mapped and simulated using the PSDS-M64.

4.6 Mapping Results and comparison

Figure 11 shows the mapping results of Dataflow Designer compared to the XPP-Vectorizing C Compiler. The rewriting logic optimized equation of the FIR filter was also implemented in the XPP-VCC in order to compare the effect of using the KressArrayXplorer for the dataflow compilation.

<table>
<thead>
<tr>
<th>XPP-VCC</th>
<th>Dataflow Designer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct form</td>
<td>Optimized (Rewriting)</td>
</tr>
<tr>
<td># ALUs</td>
<td>7</td>
</tr>
<tr>
<td># IRAMs</td>
<td>0</td>
</tr>
<tr>
<td># FREGs</td>
<td>8</td>
</tr>
<tr>
<td># BREGs</td>
<td>8</td>
</tr>
<tr>
<td>Total:</td>
<td>23</td>
</tr>
<tr>
<td>Change</td>
<td>0%</td>
</tr>
<tr>
<td>Timing Delay (cycles)</td>
<td>67</td>
</tr>
<tr>
<td>Change</td>
<td>0%</td>
</tr>
</tbody>
</table>

Figure 11. Results Comparison: Four-Tap FIR Filter

Implementing the direct form of the Four- Tap FIR filter directly in XPP-VCC offered the worst result both in resource utilization and timing delay. In contrast, the implementation by the ALEX compiler used 21% less resources with a 22% improvement in the timing delay. Applying rewriting-logic improved the results of both compilers, regarding both resource utilization and timing delay. The timing delay improvement was 28% for the XPP-VCC and 25% for the ALEX Compiler.

The results of this proof of concept illustrate how rewriting-logic can be used to improve the performance and resource utilization on coarse-grained architectures. It also demonstrates that these improvements can be obtained in an automated manner with minimum effort and analysis by the user.

5 Conclusions and future work

Dataflow Designer, a novel tool based on the combination of rewriting-logic with a retargetable dataflow compiler, was presented in this paper. By using this tool, the user can obtain alternative, correct-by-design implementations from a single mathematical specification, doing it just by selecting the rewriting-logic strategies to be applied. This alternative implementation generation process is fast.

1 It is very important to remark, that these timing delay cycles include also the cycles used for the reconfiguration of the ALU, BREG and FREG elements of the XPP.
and completely automated, allowing the user to compare, with minimum time and effort, many different implementation approaches, while avoiding human errors being introduced during the mathematical manipulation and simplification. The power of the rewriting-logic approach was illustrated by optimizing and synthesizing a Four-Tap FIR Filter. The resulting implementations used 13-21% less resources with a 25-28% improvement in the timing delay.

Future work includes the expansion of Dataflow Designer to support for different coarse-grained architectures. This will require the writing of new hardware description files for the dataflow compiler and new code converters at its output. The dataflow compiler currently used by the tool has some limitations concerning loops and recursivity which could be solved in future versions. Additionally, the compiler could also be extended to support arbitrary data types (i.e. complex) and multi-output functions. As an alternative, automated interfaces to other compilers (i.e. XPP-VCC) could also be implemented by creating the necessary code converters. Parameterized function blocks can be created with this tool and included into a parameterized library. The inclusion of these functions into the rewriting rules and hardware file will greatly increase the capabilities and usefulness of the tool. In this way, new and more complex rules and strategies can be implemented; an obvious example is a strategy to trade resource utilization for performance, based on a rule that will select a faster but bigger implementation of the submodules. Other more complex strategies could include power optimization and tradeoffs based on the available hardware resources.

Future versions of the presented tool may integrate an interface (from ELAN) to PVS [20,21,22] to allow the semi-automated verification using formal methods instead of requiring simulation. For this integration, it is necessary to include strategies for applying rewriting proving methods (such as Knuth-Bendix-Huet critical pair theorem, detection of critical pairs, verification of joinability, etc.) in PVS.

6 References

[8] P. Borovansky, C. Kirchner, H. Kirchner, and P. Moreau. "ELAN from a rewriting logic point of view". Theoretical Computer Science 285(2).