FROM EQUATION TO VHDL: USING REWRITING LOGIC FOR AUTOMATED FUNCTION GENERATION

C. Morra,[1] [*] *M. Sackmann,*[1] *S. Shukla,*[2] *J. Becker,*[1] *R. Hartenstein*[3]

[1]ITIV
Universität Karslsruhe
Karlsruhe, Germany
{morra,becker}@itiv.uni-karlsruhe.de

[2]ITEE
University of Queensland
Brisbane, Australia
sunil@itee.uq.edu.au

[3]Fachbereich Informatik
TU Kaiserslautern
Kaiserslautern, Germany
hartenst@rhrk.uni-kl.de

ABSTRACT

This paper presents a novel tool flow combining rewriting logic with hardware synthesis. It enables the automated generation of synthesizable VHDL code from mathematical equations and the quick generation of functionally equivalent alternative implementations. The simple but powerful semantics of rewriting logic provide a natural mechanism for manipulating algebraic expressions, using a high-level of abstraction which is afterwards automatically converted into lower levels of abstraction.

The design flow is validated by generating polynomial approximations for arbitrary continuous functions. The polynomial generation process is completely parameterized regarding polynomial degree, number representation parameters, word width and polynomial evaluation approaches.

Different functionally equivalent implementations for the resulting polynomial approximations were generated and synthesized for a Virtex4 device.

1. INTRODUCTION

Reconfigurable architectures are increasingly being used for digital signal processing applications. For these kind of applications, the developer interprets and manipulates set of mathematical equations and manually translates into a lower abstraction level. The developer must consider many different implementation approaches and parameters in order to obtain the best trade-offs for the given application.

The exploration of different approaches and implementation alternatives is a very complex, time consuming and error-prone process which requires a lot of expertise from the developer. To address this problem, a novel tool flow based on the combination of rewriting logic with hardware synthesis is being developed. It automatically generates synthesizable code from a mathematical specification and reduces the amount of time and effort needed to generate alternative implementations and find the best approach for the given application and target architecture.

[*]C. Morra would like to acknowledge Prof. P. Leong and Mr. K.H. Tsoi from the Chinese University of Hong Kong for providing the initial source code of the CAST system.

2. RELATED WORK

The development for reconfigurable architectures is usually done using Hardware Description Languages, which are inherently low-level. In contrast, compiler based approaches and model-based design offer a higher level of abstraction. Examples include the Handel-C compiler [1] and model-based tools like COMPAAN/LAURA [2] and Xilinx's System Generator [3].The KressArray Xplorer [4] provides a retargetable dataflow compiler and a complete system for coarse-grained hardware design space exploration.

Recent work on rewriting based treatment of hardware design includes Kapur's verification of arithmetic circuits [5] and the work from Arvind on the specification of processors with simple architectures and the rewrite-based description and synthesis of simple logical digital circuits [6]. Ayala et al. used term rewriting combined with logic strategies to model dynamically reconfigurable architectures [7][8].

3. TERM REWRITING AND REWRITING LOGIC

Term rewriting is the formal mathematical framework for the reduction of expressions using matching and substitution of terms. Term rewriting is applied in the form of rewriting rules that define how the term is transformed.

Rewriting rules are of the form:

$$s \rightarrow t \text{ if } c$$

Meaning that a sub-term that matches the left-hand side of the rule will be replaced by the right-hand side when the condition c holds. These operational semantics are the same as those involved in functional programming.

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 most adequate form to implement the associated operator or function. Thus, combining rewriting with logic strategies 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.

An example of a rewriting rule to convert a complex multiplication into real operations is:

[Rule1] *(A+i*B)*(C+i*D)* →
*((A*C-B*D)+i*(B*C+A*D)* if *A,B,C,D* are real

This is the classical form of a complex multiplication, but not the only way to compute it. Different equivalent canonical forms can be obtained by using one of the following rules (under the same condition):

[Rule2] *(A+i*B)*(C+i*D)* →
*((A*C-B*D)+i*((A+B)*(C+D)-A*C)-B*D)*

[Rule3] *(A+i*B)*(C+i*D)* → *ComplexMult(A,B,C,D)*

4. FELIX DESIGN FLOW

The *Functional Equivalent Logical Implementation eXplorer* (FELIX) [9] provides a novel design flow that combines rewriting logic with hardware synthesis, in order to quickly obtain different reconfigurable hardware (*configware*) implementation alternatives for mathematical functions. The FELIX tool flow supports both fine and coarse-grained reconfigurable architectures.

Using rewriting logic's simple semantics allows manipulating the specification on a higher abstraction level and avoids the inclusion of unnecessary operational semantics that are required in programming languages [10]. The transformed specifications are rewritten into a lower abstraction level using rewriting logic strategies that translate the operations into the operators available in the target architecture and the functions available in an external library. The resulting specifications are converted into compilable code, whose result is mapped or synthesized according to the target architecture. The design flow block diagram is shown in Fig. 1.

The rewriting is done using ELAN[11].

For coarse-grained architectures, the retargetable data-

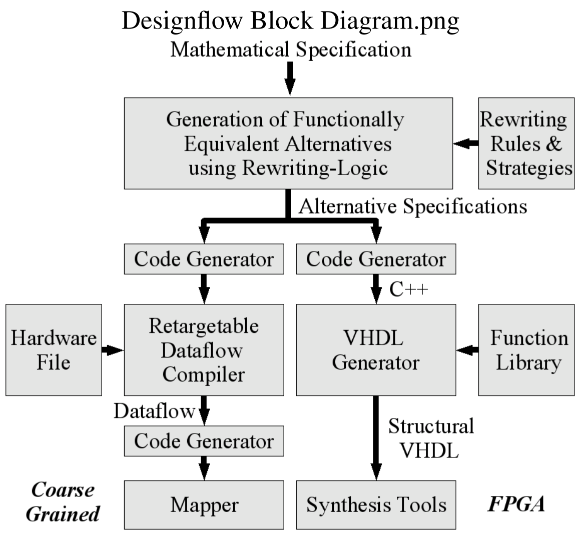
Designflow Block Diagram.png
Mathematical Specification
Generation of Functionally Equivalent Alternatives using Rewriting-Logic
Rewriting Rules & Strategies
Alternative Specifications
Code Generator
Code Generator
C++
Hardware File
Retargetable Dataflow Compiler
VHDL Generator
Function Library
Dataflow
Code Generator
Structural VHDL
Coarse Grained
Mapper
Synthesis Tools
FPGA

Fig. 1. FELIX Design Flow

flow compiler from the the KressArray Xplorer is used to generate the dataflow of the function. Mapping is done using the platform specific tools.

To generate implementations for FPGAs, the rewritten output is converted to a C++ description of interconnected function blocks which is then converted to structural VHDL, using the CAST System [12]. The resulting VHDL can be synthesized using standard synthesis tools.

Besides *configware* design space exploration, FELIX can also be used for coarse-grained *hardware* design space exploration by means of its interface to the KressArray Xplorer.

5. POLYNOMIAL APPROXIMATIONS

The only functions of one variable that can be computed using only a finite number of additions, subtractions and multiplications are polynomials. Therefore, it is sometimes useful to approximate continuous functions by polynomials [13].

A polynomial approximation is generated through an iterative process that minimizes an approximation error metric. Two commonly used metrics are the maximum error and the average error. The approximation that minimizes the maximum error is called the *minimax* approximation, while the polynomial that minimizes the average error is called the *least squares* approximation.

The average error is computed as the sum of the squares of the offsets:

$$\| p - f \|^2 = \int_a^b w(x)(f(x) - p(x))^2 dx$$

where $[a, b]$ is an interval and w a continuous weight function. The least squares polynomial p that approximates a function f is:

$$p = \sum_{i=0}^{n} a_i T_i$$

To find this polynomial, first the scalar product is defined:

$$\langle g, h \rangle := \int\limits_a^b w(x)g(x)h(x)dx$$

Then a sequence of polynomials (T_m) is found such that T_n is of degree n and which are orthogonal over $[a, b]$:

$$\langle T_m, T_n \rangle = 0 \text{ for } m \neq n$$

The coefficients a_i of the polynomial p can then be computed as follows:

$$a_i = \frac{\langle f, T_i \rangle}{\langle T_i, T_i \rangle}$$

Sets of orthogonal polynomials include the Chebyshev, Legendre and Jacobi polynomials. In this work, the Legendre polynomials in the interval $[-1, 1]$ were chosen because of their simplicity. These polynomials are the solutions to Legendre's Differential Equation:

$$(1 - x^2)\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} + n(n + 1)y = 0$$

They satisfy the following recurrence relation:

$$L_0(x) = 1, L_1(x) = x,$$

$$L_n(x) = \frac{2n-1}{n}xL_{n-1}(x) - \frac{n-1}{n}L_{n-2}(x)$$

The value of $w(x)$ is 1. The scalar products are:

$$\langle L_i, L_j \rangle = \begin{cases} 0 & \text{if } i \neq j \\ \dfrac{2}{2i+1} & \text{if } i = j \end{cases}.$$

This reduces the coefficients equation to:

$$a_i = \frac{1}{2}(2i + 1)\int_a^b f(x)L_i(x)dx$$

Integrals were approximated by trapezoid sums: given a partition of the interval $[a, b]$ into n parts ($a = x_0 < x_1 < \ldots < x_n = b$), the trapezoid sum is:

$$\frac{1}{2}\left(\frac{f(x_0)+f(x_1)}{x_1-x_0} + \ldots \frac{f(x_{n-1})+f(x_n)}{x_n-x_{n-1}}\right)$$

This means that the function f can be given simply by a set of values $(x, f(x))$, thus f can be any arbitrary continuous function.

6. POLYNOMIAL EVALUATION

It remains to consider however, how $p(x)$ is to be calculated. The first solution that comes to mind is to simply transform all powers of x to multiplications. However, there are more efficient ways to evaluate polynomials.

One possibility that is often used in computer science is Horner's rule:

$$p(x) = c_0 + x(c_1 + x(\ldots x(c_{n-1} + x * c_n) \ldots))$$

This only requires n multiplications and n additions, whereas in the previous equation the evaluation of $c_n x^n$ alone needs n multiplications.

For parallel hardware it might be even faster to use Estrin's Method [14]. Let $n = 2^k - 1$, $p_{0,i} := c_i$, $i = 0, \ldots, 2^k - 1$ and $m := x$. In the first step, the odd $p_{0,i}$ are multiplied by m and then added to the even $p_{0,i}$: $p_{1,0} = p_{0,0} + p_{0,1}x$; $p_{1,1} = p_{0,2} + p_{0,3}x$; $\ldots$; $p_{1,2^{k-1}-1} = p_{0,2^k-2} + p_{0,2^k-1}x$. Then, set $m := m * m$.

Repeat $p_{j,i} = p_{j-1,2i} + p_{j-1,2i+1}m$ and $m := m * m$ for all $j = 1 \ldots k$. The result for a 7th degree polynomial is:
$p(x) = ((c_0 + c_1 * x) + (c_2 + c_3 * x) * (x * x)) +$
$((c_4 + c_5 * x) + (c_6 + c_7 * x) * (x * x)) * ((x * x) * (x * x))$

7. IMPLEMENTATION AND RESULTS

The process described in the previous section was implemented using rewriting logic. The rewriting strategy is shown on Figure 2, where each block of the figure corresponds to a set of rewriting rules. As mentioned earlier, the input function is given as a set of points (x,f(x)) and therefore it can be any arbitrary continous function. The rewriting rules are parameterized for n (degree of polynomial), data word width, number representation (integer, fixed-point parameters) and number of input points. Three sets of rewriting rules were implemented for the evaluation of the polynomials: normal, horner and estrin. These rules are generic and work for any polynomial degree, independently of any of the previous parameters.

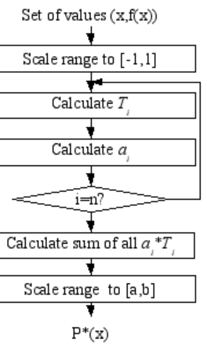

Fig. 2. Generation of Polynomial Approximations

Polynomial approximations were generated for the function $sin(exp(x))$ and $sqrt(x)$, varying the numbers of input points and the degree. Figure 3 shows some of the resulting approximations.

The hardware implementations were tested for the function $sin(exp(x))$. Two different polynomial approximations were generated: one using a 3rd degree polynomial and the

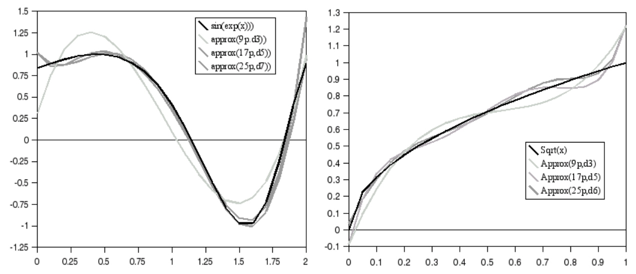

Fig. 3. Polynomial Approx.: a) $sin(exp(x))$ b) $sqrt(x)$

other using a 7th degree polynomial. Each polynomial was then implemented using three different evaluation methods: normal, Horner and Estrin. 24 bit (8.16) fixed point, 2's complement representation was used to represent the numbers. The resulting polynomials were then automatically converted to a C++ description for generating the corresponding VHDL files. The design was synthesized for the Virtex4 XC4V25LX device using the Xilinx ISE 7.1 tool.

The slice count and asynchronous delay results for the six implementations are shown in Table 1.

Table 1. Hardware Implementation Results for $sin(exp(x))$

	3rd Degree		7th Degree	
	Slices	Delay	Slices	Delay
Normal	1531 (14%)	42.50 ns	6,962 (64%)	88.50 ns
Horner	933 (8%)	45.77 ns	2,177 (20%)	92.22 ns
Estrin	1,232 (11%)	45.53 ns	2,775 (25%)	46.13 ns

In general, the Estrin algorithm provides the best timing performance while Horner offers the best resource utilization. For the 3rd degree polynomial, the timing performance of Estrin and Horner is similar, therefore Horner offers the better option. For the 7th degree polynomial, Estrin outperforms Horner by a factor of 2 at the expense of an extra 5% FPGA utilization. In this case, the best implementation depends on the application requirements.

The overall timing performance can be significantly improved by inserting pipelines. Slice count can be improved by using the available DSP blocks in the Virtex4 device.

8. CONCLUSION

The FELIX toolflow provides a complete development environment from a mathematical equation down to the hardware implementation. Its combination of rewriting logic with hardware synthesis allows the development using higher level of abstractions and the quick generation of implementation alternatives.

The capabilities of the design flow were demonstrated with the automated generation of polynomial approximations of arbitrary continous functions. The generation and transformation process is parameterized. The selection of the best resulting implementation depends on the application requirements and the polynomial degree.

9. REFERENCES

[1] Celoxica, "Handel-C for hardware design," Available at: http://www.celoxica.com/.

[2] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn Process Networks: The Compaan/Laura approach," in *Proc. Design, Automation and Test in Europe Conference (DATE'04)*, 2004.

[3] J. Hwang, B. Milne, N. Shirazi, and J. Stroomer, "System level tools for DSP in FPGAs," in *Proc. Int'l Conf. on Field-Programmable Logic and Applications (FPL'01)*, 2001.

[4] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array," in *Proc. ASP-DAC 2000*, vol. 1, 2000, pp. 163–168.

[5] D. Kapur and M. Subramaniam, "Using and induction prover for verifying arithmetic circuits," *Journal of Software Tools for Technology Transfer*, vol. 3, no. 1, pp. 32–65, Sept. 2000.

[6] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," *IEEE Micro*, vol. 19, no. 3, 1999.

[7] M. Ayala-Rincon, R. Nogueira, R. Jacobi, C. Llanos, and R. Hartenstein, "Modeling a reconfigurable system for computing the FFT in place via rewriting-logic," in *Proc. SBCCI'03*, 2003.

[8] M. Ayala-Rincon, R. Jacobi, L. Carvalho, C. Llanos, and R. Hartenstein, "Modeling and prototyping dynamically reconfigurable systems for efficient computation of dynamic programming methods by rewriting-logic," in *Proc. SBCCI'04*, 2004.

[9] C. Morra, J. Becker, M. Ayala-Rincon, and R. Hartenstein, "FELIX: Using rewriting-logic for generating functionally equivalent implementations," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2005)*, vol. 1, Aug. 2005, pp. 25–30.

[10] N. Marti-Oliet and J. Meseguer, Eds., *Special issue on Rewriting Logic and its Applications.*, ser. Theoretical Computer Science, 2002, vol. 285, no. 2.

[11] P. Borovansky, C. Kirchner, H. Kirchner, and P. Moreau, "Elan from a rewriting logic point of view," *Theoretical Computer Science*, vol. 285, no. 2, 2002.

[12] K. Tsoi, "Computer Arithmetic Synthesis Technologies on reconfigurable platforms," in *Proc. International Conference on Field Programmable Logic and Applications (FPL 2005)*, vol. 1, Aug. 2005, pp. 713–714.

[13] J.-M. Muller, *Elementary Functions: algorithms and implementation.* Birkhaeuser Boston, 1997.

[14] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2.