

USING REWRITING LOGIC TO GENERATE DIFFERENT IMPLEMENTATIONS OF POLYNOMIAL APPROXIMATIONS IN COARSE-GRAINED ARCHITECTURES

C. Morra,¹ M. Sackmann,¹ J. Becker,¹ R. Hartenstein²

¹ITIV
Universitaet Karlsruhe
Karlsruhe, Germany
{morra,becker}@itiv.uni-karlsruhe.de

²Fachbereich Informatik
TU Kaiserslautern
Kaiserslautern, Germany
hartentst@rhrk.uni-kl.de

ABSTRACT

A novel toolflow based in rewriting-logic is used to automatically generate polynomial approximations for arbitrary continuous functions. The simple but powerful semantics of rewriting logic provide a natural mechanism for manipulating algebraic expressions, allowing the development to be done on a higher abstraction level while avoiding the unnecessary semantics required in hardware description and programming languages. The resulting polynomial approximations are rewritten to generate alternative implementation approaches which are automatically converted into different functionally equivalent hardware implementations. The rewriting-logic toolflow can generate implementations for both fine- and coarse-grained architectures. This paper presents the implementation results for the Pact XPP coarse-grained reconfigurable architecture.

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]. Reconfigurable arrays can be classified by level of reconfiguration capability in fine- and coarse-grained. A coarse-grained array is defined as an array of functional elements with path-widths greater than 1 bit; In other words, reconfiguration is done at the function element level instead of the gate level.

Reconfigurable architectures are increasingly being used for digital signal processing applications. The typical development process for DSP applications starts with a set of mathematical equations which are manipulated and interpreted by the developer, and then manually translated 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 on the target architecture.

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 rewriting logic is being developed.

2. RELATED WORK

Diverse tools and approaches have been used to program coarse-grained architectures. The Pact XPP³ coarse-grained array is programmed using the low-level Native Mapping Language(NML) from the Pact Software Development Suite. A higher level of abstraction is offered by the Pact XPP-VCC Compiler [3] which uses automated loop unrolling loop vectorization to parallelize a C program for the XPP.

Compiler-based approaches have been popular because of the familiarity of developers with traditional programming languages. An 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 compiler for different coarse-grained architectures. An example of such retargetable compiler is the DRESC Compiler [4]. The KressArray Xplorer [5][6] provides a retargetable dataflow compiler and a complete system for hardware design space exploration.

Recent work on rewriting based treatment of hardware

³ The Pact XPP [2] is an array of coarse-grain, data-driven, runtime reconfigurable ALU and RAM elements. The ALUs can perform arithmetic and logical functions, including addition, subtraction, multiplication, shifting, AND, OR, complex addition and multiplication, swap, merge, demux and other specialized functions. Each ALU Processing Array Element has also a Forward Register (FREG) and Backward Register (BREG) that are used for routing. Additionally a FREG can do shift operations while a 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 XPP-64A, 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.

design includes the work from Kapur, who used the well-known Rewriting Rule Laboratory - RRL for verifying arithmetic circuits [7]. Arvind applied rewriting in the specification of processors with simple architectures, the rewrite-based description and synthesis of simple logical digital circuits and the description of cache protocols over memory systems [8]. 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. In this context, Ayala et al. used rewriting logic for modelling dynamically reconfigurable architectures [9][10]. To the authors' knowledge, there is no toolflow using rewriting-logic that generates actual implementations on reconfigurable hardware.

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 environments and have been promoted in functional programming languages since the well-known McCarthy LISP of the 1950s[11].

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:

$$[\text{Rule1}] (A+i*B)*(C+i*D) \rightarrow (A*C-B*D)+i*(B*C+A*D) \quad \text{if } A,B,C,D \text{ are real}$$

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):

$$[\text{Rule2}] (A+i*B)*(C+i*D) \rightarrow ((A*C-B*D)+i*((A+B)*(C+D)-A*C)-B*D)$$

$$[\text{Rule3}] (A+i*B)*(C+i*D) \rightarrow \text{ComplexMult}(A,B,C,D)$$

The most popular rewriting logic computational environments are Maude[12], ELAN[13] and CafeOBJ[14].

4. FELIX DESIGN FLOW

The *Functional Equivalent Logical Implementation eXplorer* (FELIX)[15] 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 [16]. 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.

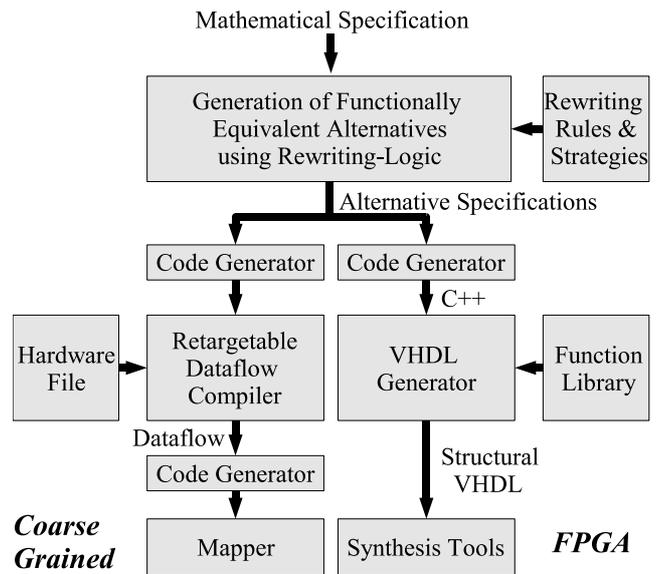


Fig. 1. FELIX Design Flow

To generate implementations for FPGAs, the rewritten output is translated to a C++ description of interconnected function blocks which is then converted to structural VHDL using the CAST System [17]. The resulting VHDL can be synthesized using standard synthesis tools.

For coarse-grained architectures, the rewritten output is converted to ALEX code for the KressArray Xplorer, whose retargetable dataflow compiler allows the generation of implementations for different target architectures by changing the target hardware description in the hardware file. The Hardware File for the XPP architecture consists of the definition of the arithmetical and logical operations of the ALU blocks and external modules. The output of the compiler is a dataflow file consisting of a list of linked operator nodes and/or functions. The operator nodes correspond to an XPP ALU, FREG or BREG, while function nodes correspond to external modules. The resulting dataflow is converted to NML code for mapping. The mapping and simulation is done using the Pact Software Development Suite.

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

The correctness of the rewriting rules can be verified through proof assistants, using the interface and methodologies described in [18]. When the rewriting rules are correct, the generated alternatives should be, by definition, "correct-by-design".

5. POLYNOMIAL APPROXIMATIONS

The only functions of one variable that can be computed using only a finite number of additions, subtractions, multiplications and comparisons are polynomials. Therefore, it is sometimes useful to approximate continuous functions by polynomials [19].

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:

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

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:

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

They satisfy the following recurrence relation:

$$L_0(x) = 1$$

$$L_1(x) = x$$

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

The weight function $w(x)$ is 1. The values of the scalar products are:

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

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 ($a = x_0 < x_1 < \dots < x_n = b$), the trapezoid sum is:

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

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, so e.g. $c_0 + c_1 * x^1 + c_2 * x^2 + \dots + c_n * x^n$ becomes:

$$p(x) = c_0 + c_1 * x + c_2 * x * x + \dots + c_n * x * \dots * x$$

However, there are more efficient ways to evaluate polynomials.

6.1. Horner's Scheme

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

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

For a degree n polynomial, this evaluation form requires n multiplications and n additions, whereas in the previous equation the evaluation of $c_n x^n$ alone needs n multiplications.

6.2. Estrin's Method

For parallel hardware it might be even faster to use Estrin's Method [20].

Let $n = 2^k - 1$, $p_{0,i} := c_i$, $i = 0, \dots, 2^k - 1$ and $m := x$. In the first step, the odd $p_{0,i}$ are multiplied by m and then added to the even $p_{0,i}$: $p_{1,0} = p_{0,0} + p_{0,1}x$; $p_{1,1} = p_{0,2} + p_{0,3}x$; \dots ; $p_{1,2^{k-1}-1} = p_{0,2^{k-2}} + p_{0,2^k-1}x$. 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 $j = 1 \dots k$.

The resulting Estrin form for a 7th degree polynomial can be seen in Figure 2.

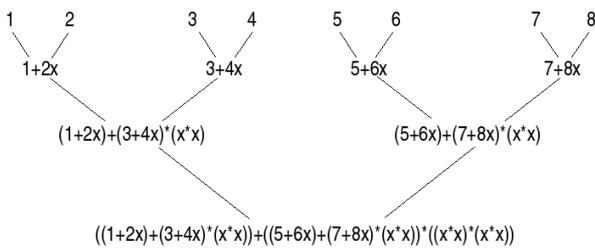


Fig. 2. Estrin form for a 7th degree polynomial

7. IMPLEMENTATION AND RESULTS

The process described in the previous section was implemented using rewriting logic in the ELAN environment. The rewriting strategy is shown on Figure 3, where each block of the figure corresponds to a set of rewriting rules.

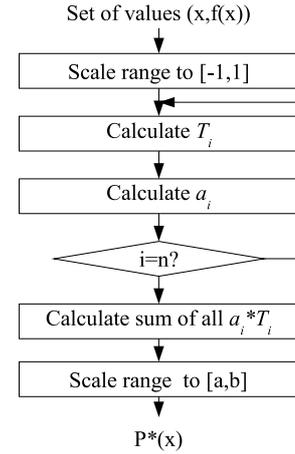


Fig. 3. Generation of Polynomial Approximations

As mentioned earlier, the input function is given as a set of points $(x,f(x))$ and therefore it can be any arbitrary continuous 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.

Polynomial approximations were generated for the functions $exp(-x^2)$, $sin(exp(x))$, $sqrt(x)$ and $tan(1.5x)$, varying the numbers of input points and the degree. Figures 5 and 4 show the resulting approximations.

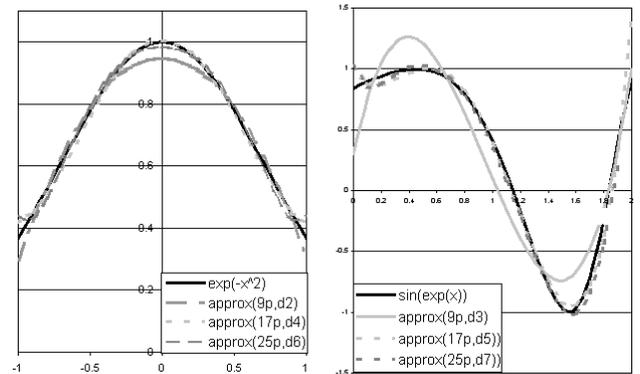


Fig. 4. Polynomial Approx.: a) $exp(-x^2)$ b) $sin(exp(x))$

The hardware implementations were tested in the XPP for a 7th polynomial. An integer and fixed-point version of the polynomial were generated and each version was then transformed using three different evaluation methods: normal, Horner and Estrin. The resulting polynomials were then automatically converted to the Pact NML language.

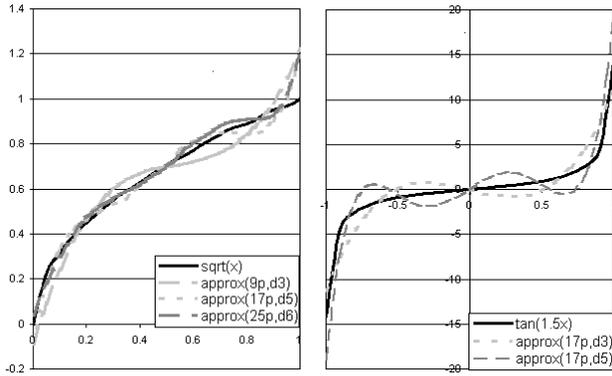


Fig. 5. Polynomial Approx.: a) \sqrt{x} b) $\tan(1.5x)$

Mapping and Simulation was done using the Pact Software Development Suite for the XPP64A device.

The number of operations and mapping results are presented in Table 1 and Table 2. The timing results are shown in Table 3.

Table 1. Number of Operations for a 7th degree Polynomial

Implementation	# ADD	#MUL	#SHIFTR
Normal	7	28	0
Horner	7	7	0
Estrin	7	12	0
Fixed-Point Horner	7	7	7
Fixed-Point Estrin	7	12	12

Table 2. Mapping results for a 7th degree Polynomial

Implementation	# ALUs	#FREGs	#BREGs
Normal	35	11	15
Horner	14	6	8
Estrin	19	9	1
Fixed-Point Horner	14	10	10
Fixed-Point Estrin	19	15	12

As expected, the Horner form used the least number of hardware resources, followed by the Estrin form. The difference between the integer and the fixed-point implementations was the extra shift-right step for every multiplication that is required to maintain the number of fractional bits. The mapper placed each shift right operation in a FREG, thus avoiding the use of extra ALU resources.

The timing results are interesting because at first glance they seem counterintuitive. All three evaluation approaches differ only in the number of cycles required for the first

Table 3. Timing results for a 7th degree Polynomial

Implementation	Clock Cycles	
	First result	Subsequent
Normal	164	1
Horner	80	1
Estrin	100	1
Fixed-Point Horner	112	1
Fixed-Point Estrin	148	1

output and there is no timing difference for subsequent values. The reason for this is simple: the communication between the blocks inside the XPP is data-driven, thus implicitly pipelined. The long initial delay is a result of the XPP (re)configuration which is done at runtime.

In this case, the best polynomial evaluation approach for the XPP architecture is clearly the Horner form. It uses the least number resources, the fastest configuration/pipeline-fill time and outputs a value each clock cycle.

8. CONCLUSION

The FELIX toolflow provides a complete development environment from a mathematical equation down to the hardware implementation. The use of rewriting logic 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 to arbitrary continuous functions. The resulting approximations were rewritten for three different polynomial evaluation approaches and integer/fixed-point implementations. The results were mapped in the XPP-64A coarse-grained array. All three approaches produced a result each clock cycle once the pipeline was filled. The best polynomial evaluation approach for the XPP is the Horner scheme because it uses fewer resources and provides the fastest runtime configuration delay between the three approaches.

The XPP's internal communication model and configuration management are very different from the ones used in FPGAs, therefore, an approach that offers the best performance in an FPGA (e.g. Estrin's method) may produce sub-optimal results in the XPP.

9. REFERENCES

- [1] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Design, Automation and Test in Europe Conference (DATE'01)*, 2001.
- [2] V. Baumgarten, G. Ehlers, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt, "PACT XPP - a self-reconfigurable data

- processing architecture,” *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.
- [3] J. Cardoso and M. Weinhardt, “XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture,” in *Proc. 12th Int’l Conf. on Field-Programmable Logic and Applications (FPL’02)*, 2002.
- [4] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwe-reins, “DRESC: A retargetable compiler for coarse-grained reconfigurable architectures,” in *Proc. Int’l Conference on Field Programmable Technology (FPL’02)*, 2002.
- [5] R. W. 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.
- [6] U. Nageldinger, “Coarse-grained reconfigurable architectures design space exploration,” Ph.D. dissertation, Universität Kaiserslautern, Germany, 2001.
- [7] 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.
- [8] Arvind and X. Shen, “Using term rewriting systems to design and verify processors,” *IEEE Micro*, vol. 19, no. 3, pp. 36–46, 1999.
- [9] 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.
- [10] 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.
- [11] H. Kirchner and P. Moreau, “Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories,” *Journal of Functional Programming*, vol. 11, no. 2, pp. 32–65, 2001.
- [12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, “Maude: specification and programming in rewriting logic,” *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.
- [13] 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.
- [14] R. Diaconescu and K. Futatsugi, “Logical foundations of CafeOBJ,” *Theoretical Computer Science*, vol. 285, no. 2, pp. 289–318, 2002.
- [15] 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.
- [16] N. Martí-Oliet and J. Meseguer, Eds., *Special issue on Rewriting Logic and its Applications.*, ser. Theoretical Computer Science, 2002, vol. 285, no. 2.
- [17] 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.
- [18] T. M. Sant’Ana and M. Ayala-Rincon, “Verification of rewrite based specifications using proof assistants,” in *Proc. XXXI Conferencia Latinoamericana de Informatica (CLEI 2005)*, 2005, pp. 699–710.
- [19] J.-M. Muller, *Elementary Functions: algorithms and implementation*. Birkhaeuser Boston, 1997.
- [20] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1998, vol. 2.