Generation of Distributed Arithmetic Designs for Reconfigurable Applications

Christophe Bobda, Ali Ahmadinia, Jürgen Teich
University of Erlangen-Nuremberg
Department of computer science
Am Weichselgarten 3, 91058 Erlangen, Germany

Klaus Danne
University of Paderborn
Heinz Nixdorf Institute
Fürstenallee 11, 33102 Paderborn, Germany

Abstract: We present a tool for design and implementation of reconfigurable computing applications based on the use of distributed arithmetic. Our tool provides the user the possibility to investigate different tradeoffs like area vs speed for his design. After simulation of the design, a synthesizable HDL code for a reconfigurable platform can be generated. Beside the existing fixed-point solutions for real numbers, we present a new approach to handle real numbers in the IEEE 754 floating-point format. The tool is used in the implementation of two applications. The first one is the implementation of a recursive convolution algorithm for time domain simulation of multimode intrasystem interconnects and the second one is the implementation of adaptive mechatronical multi-controller systems.

1 Introduction

Distributed arithmetic (DA) [Wh89, Mi00, Xi95, Xi00a] is usually defined as computation using look-up table. The main application of DA is the dot-product computation of two vectors, where one of the two vectors is constant (i.e all the elements are constant values). In this case, all the additions in which at least one element of the constant vector is involved are precomputed and stored in a look-up table. At run-time, the elements of the variable vector are used to address the look-up table and retreive partial sums in a bit-serial (1-BAAT, 1 Bit At A Time) manner.
One of the notable contribution in DA has been done by White [Wh89]. He proposed the use of ROMs to store the pre-computed values. The surrounding logic to access the ROM and retrieve the partial sums has to be implemented on a separate chip. Because of this moribund architecture, the DA method could not be successfully used. With the appearance of SRAM based FPGAs, the DA became an interesting alternative to implement signal processing application in FPGA [Mi00, Xi95, Xi00a]. Because of the availbility of SRAMs in those FPGAs, the precomputed values could now be stored in the same chip as the surrounding logic.

Altough DA is used for many applications, the users still have to design their systems and investigate the different tradeoffs by hand. This process is not always easy and can be time consuming. On the other hand, fixed-point format is used to represent real numbers. This results in the lost of accuracy as well as the limitation of the numbers range. We have developed a framework to help designers in the development of signal processing applications using the DA. Moreover we are able to handle real number in the IEEE 754 floating-point format.
The rest of the paper is organized as follows: Section 2 provides the basics of distributed arithmetics as well as our method for handling real numbers. In section 3 our tool, the DA generator is presented while section 4 shows how DA can efficiently benefit from partial reconfiguration. In section 5 we present the application of our framework for the generation and evaluation of the code for two signal processing applications. Section 6 concludes the paper and gives some indications for future work.

2 Distributed Arithmetic

The idea behind the distributed arithmetic is to distribute the bits of one operand across the equation to be computed, such as to obtain a new equation which can then be computed in a more efficient way. For the dot-product, we are given the following equation:

$$Z = X \times A = \sum_{i=0}^{n} (X_i \times A_i) \tag{1}$$

The vector A is a constant vector of dimension n while X is a variable vector of the same dimension. With the binary representation $\sum_{j=0}^{w-1} X_{ij} 2^j$ (where w ist the width of the variables and $X_{ij} \in \{0,1\}$ is the j-th bit of X_i) of the X_is, equation (1) can be written as:

$$Z = \sum_{i=0}^{n} A_i \times \sum_{j=0}^{w-1} X_{ij} 2^j = \sum_{j=0}^{w-1} 2^j \sum_{i=0}^{n} X_{ij} A_i \tag{2}$$

Equation (2) represents the general form of a distributed arithmentic since each bit of each variable operand contributes only once to the sum $\sum_{i=0}^{n} X_{ij} A_i$. Because $X_{ij} \in \{0,1\}$, the number of possible values $\sum_{i=0}^{n} X_{ij} A_i$ can take is limited to 2^n. Therefore, they can be precomputed and stored in a look-up table. We call such a look-up table a distributed arithmetic look-up table (DALUT). The DALUT has the size $w \times 2^n$ bits. The n-tupel $(X_{1j}, X_{2j}, ..., X_{nj})$ is then used to address the DALUT and retrieve the value $\sum_{i=0}^{n} X_{ij} A_i$. The complete dot-product Z requires w steps to be computed in a 1-BAAT manner. At step j, the j-th bit of each variable are used to address the DALUT and retrieved the value $\sum_{i=0}^{n} X_{ij} A_i$. This value is then left shifted by a factor j (which corresponds to a multiplication by 2^j) and accumulated. After w accumulations, the values of the dot-product can be collected. The DA datapath is then obvious as illustrated in Figure 1. Many enhancements can be done on a DA implementation. The size of the DALUT for example could be halved if only positive values are stored. In this case, the first bits of a number representing the sign will be used to decide if the retrieved value should be added

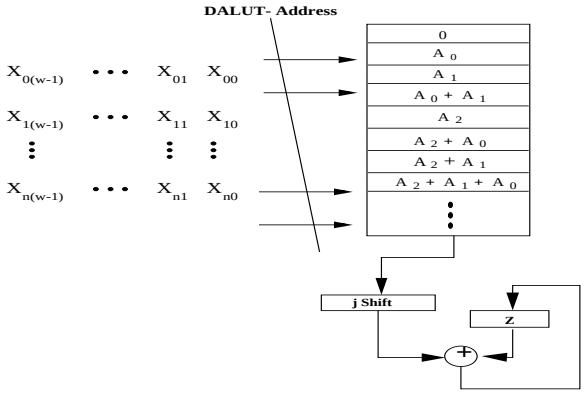

Abbildung 1: The DALUT dot-product computation

or substracted from the accumulated sum. On the other hand it is obvious that all the bit operations are independent from each other and therefore could be done in parallel. The degree of parallelism depends on the available memory to implement the DALUTs. In the case where w DALUTs can be instantiated in parallel, computation of the complete dot-product can be done in only one step. In general, if k DALUTs are instantiated in parallel, then w/k steps are required for the complete computation. The computation is done in this case on a k-BAAT basis.

2.1 High Dimensional Distributed Arithmetic

In many areas, for example in mechanical control, computations are not only limited to dot-product. Matrix operations are used as shown in equation (3).

$$\begin{pmatrix} z_1 \\ z_2 \\ ... \\ z_s \end{pmatrix} = \begin{pmatrix} a_{11} & ... & a_{1r} \\ & ... & \\ a_{1s} & ... & a_{sr} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_r \end{pmatrix} \tag{3}$$

Equation (3) can be implemented using s DALUTs. The i-th ($i \in \{1,..,s\}$) DALUT is used for the dot product $z_i = \sum_{j=0}^{r} x_j \times a_{ij}$ using the constants a_{i1} to a_{ir}. If there is enough space on the chip to hold all the DALUTs and the resulting adder tree, then equation (3) can be computed in just one clock. If there is not enough space to hold the adder tree but enough memory to hold the DALUTs then the computation have to be done sequentially. As we will see later partial reconfiguration can also be used to implement many dot-products if there is not enough memory on the chip to hold all the DALUTs.

2.2 Floating-point Distributed Arithmetic

The straightforward approach to handle real numbers in distributed arithmetic is the use of fixed-point which does not differ from the integer implementation. However, the range of a fixed-point representation as well as their precision is small compared to those of a floating-point representation. Therefore we will like to handle real numbers as floating-point. In this section we present our concept for handling real numbers as floating-point in the IEEE 754 format in distributed arithmetic. In IEEE 754 format, a number X is represented as follows:

$$X = (-1)^{S_X} 2^{e_X} \times 1.m_X \tag{4}$$

Where e_X is the exponent (we consider that the substraction with the bias is done and the result is e_X), m_X is the mantissa and S_X is the sign of X. Without loss of generality, we will consider the sign to be part of the mantissa, thus the new representation is $X = 2^{e_X} \times m_X$. With A, X and Z being all floating-point numbers, the floating-point counterpart of equation (1) is given by:

$$\begin{aligned} Z = X \times A = \sum_{i=0}^{n} (X_i \times A_i) = \sum_{i=0}^{n} (2^{e_{A_i}} \times m_{A_i}) \times (2^{e_{X_i}} \times m_{X_i}) \\ = \sum_{i=0}^{n} (2^{e_{A_i}} \times 2^{e_{X_i}}) \times (m_{A_i} \times m_{X_i}) = \sum_{i=0}^{n} (2^{e_{A_i}+e_{X_i}}) \times (m_{A_i} \times m_{X_i}) \end{aligned} \tag{5}$$

Our goal is to compute and provide Z as floating-point number. Therefore we would like to read (at each step of the computation) a floating-point value F_i from the floating-point DALUT and add it to an accumulated sum which is also a floating-point number. Since the adder used at this stage is a floating point adder, issues like rounding and normilization will be considered in its implementation. From the last part of equation (5), it is obvious that the value $(2^{e_{A_i}+e_{X_i}}) \times (m_{A_i} \times m_{X_i})$ represents a floating-point number with exponent $e_{A_i} + e_{X_i}$ and mantissa $m_{A_i} \times m_{X_i}$. By setting $e_{F_i} = e_{A_i} + e_{X_i}$ and $m_{F_i} = m_{A_i} \times m_{X_i}$, we have the requested values at each computation step. Instead of computing the exponential part $(2^{e_{A_i}+e_{X_i}})$ of F_i as well as its mantissa $(m_{A_i} \times m_{X_i j})$ online, our approach consist of using two floating-point DALUTS for each constant A_i. We precompute and save the values $e_{A_i} + e_{x_i}$ in the first DALUT and $m_{A_i} \times m_{X_i}$ in the second one. We call the first DALUT which stores the exponents the EDALUT and the second DALUT which stores the mantissas the MDALUT. The size of EDALUT: $size(EDALUT)$ as well as that of MDALUT: $size(MDALUT)$ are defined in equations (6).

$$size(EDALUT) = n \times 2^{|E|} \times |E| \text{ bits} \tag{6}$$

$$size(MDALUT) = n \times 2^{|M|} \times |M| \text{ bits} \tag{7}$$

$|E|$ is the exponent width and $|M|$ is the mantissa width of the floating-point representation. The main argument against our approach could be that the size of the DALUTs used for this floating-point DA implementation will be too big and therefore, the method could not

be implemented. But if we consider a DA implementation involving five variables and five coefficients represented in the IEEE 754 floating-point format with 8 bits exponent and 10 bits mantissa, the total memory requirement for the EDALUTs and the MDALUTs is: $((5 \times 2^8 \times 8) + (5 \times 2^{10} \times 10))/1024 = 60$ Kbits. Therefore the EDALUTS and the MDALUTs will easily fit into the smallest low cost FPGA (the Spartan III, 50) from Xilinx which has 72 Kbits Block RAM [Xi00b]. Our approach is therefore suitable for FPGA implementation.

Having the EDALUTs and the MDALUTs for each coefficient, the datapath will not

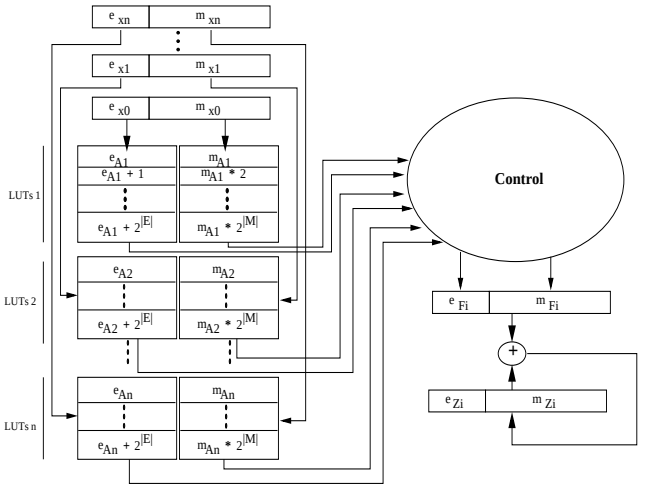

Abbildung 2: The DALUT dot-product computation

be implemented as in the fixed-point or integer case. The variables are no more input in a bit-serial way. At step i the variable X_i is used to address the two i-th EDALUT and MDALUT. The bits of e_{X_i} are used to access the EDALUT while the bits of m_{X_i} are used to access the MDALUT in parallel. The values collected from the EDALUT and the MDALUT are used to build the floating point number F_i. After n steps the floating-point dot-product is computed. Figure 2 shows the datapath for the floating-point DA. Since all the DALUTs for the n coefficients are available on the device, they can be accessed in parallel to compute the dot-product in only one step.

3 The DA Generator

We have developed a tool to help the user in the development of DA-based signal processing applications. Because those applications are usually based on the computation of the dot-product of one vector of variables with a vector of constants, they are ideal candidates for a DA implementation in reconfigurable hardware. In our tool, the user can investigate the different tradeoffs for his application. For a given device and a given set of constants, we generate different DA implementations from the full adder tree (w-BAAT) which can

be computed in only one step to the full sequential DA (1-BAAT). For each possibility, the user is provided the area and speed of the design. Real numbers can be handled either as fixed-point or as floating-point in the IEEE 754 format with the technique previously defined. The width of the mantissa as well as that of the exponent has to be provided. As Hardware Description Language we use the Handel-C language.

4 Use of Reconfiguration

The possibility of exchanging mechatronical controllers in a running system using FPGA has been presented in [DBK03]. An adaptive mechatronical system made upon a monitoring part remaining fix (FM) all the time and two reconfigurable controllers (CM1, CM2) are presented. At each point of time, one of the two controllers must be active to control the plant. Depending on the environment factors, the monitoring part can place a new controller on a predefined slot occupied by the inactive module using partial reconfiguration. The controller of the plant is then changed by switching from the active controller to the new loaded one. With this, the system remains active while adapting itself to its environement. The controller codes are implemented and stored as partial bitstreams in a ROM and can be downloaded in the corresponding slot by the monitoring module. A substantial effort have been done to implement the system in FPGA. Because the routing tools can route the connection between the fixed module and the reconfigurable controller in different ways across different configurations, the system may not work properly if partial reconfiguratioin is done to replace an old controller. To avoid this, fixed communication channels had to be defined to allow the fix monitoring module to communicate with the reconfigurable parts of the designs. This process is difficult to be automatized with the current tools. Therefore most of the work has to be done by hand.
In general, reconfiguration can be done either by changing the DALUTs content or by changing the datapath from a x-BAAT to a y-BAAT implementation. The previous described adaptive mechatronical system can be easily implemented using distributed arithmetic. Moreover, for a x-BAAT datapath, the control part is the same for all DA designs with a given number of variables, it does not need to be changed on reconfiguration. The only thing we need to do is to fill the memory with the correct combinations of the constant values and restart the computation with the fixed control part. This process is easy to be automatized and can be integrated in CAD tools.

5 Applications

Our tool was used in the design of two signal processing applications with different tradeoffs. The first application is the recursive convolution algorithm of time domain simulation of optical multimode intrasystem interconnects and the second is the implementation of the adaptive mechatronic controller previously described. The two applications are described below.

5.1 Recursive convolution algorithm of time domain simulation of optical multimode intrasystem interconnects

In general, an optical intrasystem interconnect contains several receivers with optical inputs driven by transmitters with optical outputs. The interconnections of transmitter and receivers are made by a passive optical waveguide which are represented as a multiport (figure 3) using ray tracing approach. The transfer of an optical signal along the wave-

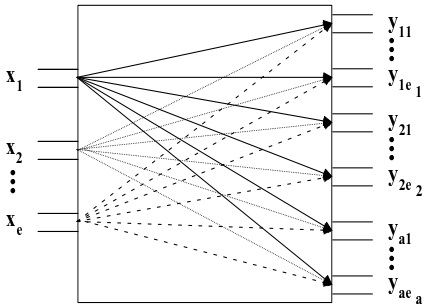

Abbildung 3: An optical multimode waveguide is represented by a multiport with several transfer paths.

guide can be computed by a multiple convolution process. Frequency domain simulation methods are not applicable regarding to the high number of frequency. Pure time domain simulation methods are more efficient if the pulse responses can be represented by exponential functions.The application of a recursive method for the convolution of the optical stimulus signals at the input ports with the corresponding pulse responses enables a time efficient computation of the optical response signals at the belonging output ports. The recursive formula to be implemented in three different intervals is given by equation (8).

$$\begin{aligned} y(t_n) &= f_0 \cdot y(t_{n-1}) + \\ & \quad f_4 \cdot x_0 - f_5 \cdot x_1 + f_{24} \cdot x_2 + f_{53} \cdot x_3 \,. \end{aligned} \tag{8}$$

f_0, f_4, f_4, f_{24} and f_{53} are constants while t_{n-1}, x_0, x_1, x_2, x_3 are variables. Therefore DA can be applied for this computation.
For this equation, different tradeoffs were investigated in our framework. A handel-C code were generated and the complete design was implemented on a system made upon the Celoxica RC100-PP board equipped with a Xilinx Virtex 2000E FPGA and pluged into a workstation.

The workstation is used for sending the variable to the FPGA and collecting the result of the computation. The implementation of equation (8) in three intervals occupies about 14 % of the FPGA area while running at 65 MHZ. Because we had no restriction on the space, our goal was to implement the maximum *k*-parallel DALUT and therefore, increase the computation speed. We could therefore implement a 6-parallel level DALUT, thus increasing the performance of factor 6. The 6-parallel DALUT and the corresponding adder three occupy about 76 % of the FPGA area. Enough space is left for the routing.

Workstation	**1 interval**	**3 intervals**
Sun Ultra 10	73.8 ms	354.2 ms
Athlon (1.53 GHZ)	558 ms	1967.4 ms
FPGA (time)	**1 interval**	**3 intervals**
Pure dot-product	25.6 ms	76.8 ms
Sequential DA	19.4 ms	19.4 ms
3-parallel DA	6.4 ms	6.4 ms
FPGA (area)	**1 interval**	**3 intervals**
	fit	fit
Pure dot-product	no	no
Sequential DA	yes (7 %)	yes (14 %)
3-parallel DA	yes (14 %)	yes (42 %)

Tabelle 1: Results of the recursive convolution equation on different plattforms

The same design was implemented without use of DA. It could not fit into the same FPGA and the run-time was much longer. The comparison of our DA implementation with other implementation is given in table1 . The performance as well as the area consumption of our implementation is more efficient than that of all the other architectures.

5.2 Digital Linear Controller

The task of a controller is to influence the dynamic behavior of a system referred as *plant*. If the input values for the plant are calculated on basis of the plant's outputs, we refer to a control feedback.
A common basic approach is to model the plant as a linear time-invariant system. Based on this model and the requirements of the desired system behavior, a linear controller is systematically derived using formal design methods. The controller as a result of the synthesis considered above, is described as a linear time-invariant system and a time discretization is performed which results in equation (9). The input vector of the controller is represented by **u** (measurements from sensors of the plant),**y** is the output vector of the controller (regulating variable to actuators of the plant) and **x** is the inner state vector of the controller. The matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and **D** are used for the calculation of the outputs based on the inputs.

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \\ \mathbf{y}(k) &= \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \end{aligned} \tag{9}$$

where $p = dim(\mathbf{u})$, $n = dim(\mathbf{x})$ and $q = dim(\mathbf{y})$

The task of the digital system is to calculate equation (9) during one sampling interval. That includes determining the new state $\mathbf{x}_{k+1}$ and the output $\mathbf{y}_k$ before the next sampling point $k+1$.

The state space equations of a digital linear time invariant controller (equation (9)) can be

written as a product of a matrix of fix coefficients and a vector of variables.

$$\mathbf{z}_{(n+q,1)} = \mathbf{M}_{(n+q,n+p)}\mathbf{v}_{(n+p,1)} \tag{10}$$

In general, the calculation of $\mathbf{z}$ evolves $n+q$ times the computation of an $n+p$ dimension scalar product (each row of $\mathbf{M}$ has to be multiplied with $\mathbf{v}$). The matrix $\mathbf{M}$ is often sparse, thus the dimension of the scalar products can be reduced from $n+p$ to e_i, where e_i is the number of non zero places in row i of $\mathbf{M}$. This can be used to reduce the size of the DALUT in a DA implementation of eq. 10.

5.3 Tradeoffs of Distributed Arithmetic Implementation

Equation (10) was implemented in our framework and the area/time tradeoff was explored on different levels from the 1-BAAT to the w-BAAT computation. Further on, we have investigated the different levels of parallelism for the $n+q$ dimensionality. This ranges from the full sequential implementation of the involved dot-products using one data-path to the full parallel implementation using $n+q$ data-paths. Modern FPGA architetures offer several techniques to implement the DALUTs. For each dot-product block-RAM can be used. If we consider using dual-port RAM, then the scalar-products can be computed in a 2-BAAT fashion. Because the DALUT entries do not change during the computation, higher order multi-port RAMs can be used. Those RAMs easily can be builded by multiple instantiation of one RAM. Finaly, the DALUT can be implemented using the FPGA look-up tables.
The controller which was used in this experiment is that of the inverse pendulum. It has three inputs ($p=3$), two inner states ($n=2$) and one output ($q=1$) and operates with a wordwidth of 20 bits. Therefore the matrix $\mathbf{M}$ has a dimension of $(3,5)$. Since some of the matrix entries are zero, three scalar-products have to be computed with dimension 2, 2 and 5 respectively. Table 2 shows the synthesis results for trade-offs using 1-BAAT (full sequential), 5-BAAT, 10-BAAT and 20-BAAT (full parallel). It shows the latency as well as the area occupation (as number of slices) of the different implementations on an FPGA VirtexE.

Archit.	Cycles	Latency (ns)	Datapath (op.inputs)	Area (slices)	AT-Product (slices*ns)
1-BAAT	20	645	2	669	431505
2-BAAT	10	344	3	767	263641
5-BAAT	4	159	6	1022	162817
10-BAAT	2	93	11	1379	128749
20-BAAT	1	45	21	1715	76727

Tabelle 2: Invers pendulum controller: Synthesisresults

6 Conclusion

We have presented a tool for design and implementation of signal processing applications based on the use of distributed arithmetic. After a short introduction of the distributed arithmetic, we have shown how to handle real numbers in the IEEE 754 floating-point format. We have used the the framework for the implementation of two signal processing applications as well as the investigation of different trade-offs. The results were presented and comparison to other implementation was also made. Future work includes the introduction of partial reconfiguration routine in the framework. This will help the user to easily mark the part of his application which will be replaced at run-time. The framework will then be used to generate the partial configuration which will be used to move from one state of the device to the next one.

Literatur

[DBK03] Danne, K., Bobda, C., und Kalte, H.: Run-time exchange of mechatronic controllers using partial hardware reconfiguration. In: *Proc. of the International Conference on Field Programmable Logic and Applications (FPL2003), Lisbon, Portugal*. September 2003.

[Mi00] Minzer, L.: Programmable silicon for embedded signal processing. *Embedded Systems Programming*. S. 110–133. March 2000.

[Wh89] White, S. A.: Application of distributed arithmetic to digital signal processing: A tutorial review. *IEEE ASSP Magazine*. S. 4–19. July 1989.

[Xi95] Xilinx: A guide to using field programmable gate arrays (fpgas) for application-specific digital signal processing performance. *http://www.xilinx.com*. 1995.

[Xi00a] Xilinx: The role of distributed arithmetic design in fpga-based signal processing. *http://www.xilinx.com*. 2000.

[Xi00b] Xilinx: Spartan-3 fpgas. *http://www.xilinx.com*. 2000.