Dataflow-Based Reconfigurable Architecture for Streaming Applications

Anja Niedermeier, Jan Kuper, Gerard Smit
University of Twente, Department of Computer Science
Enschede, The Netherlands

***Abstract*—Coarse-grain reconfigurable arrays often rely on an imperative programming approach including a read/write mechanism for memory access. In this paper, we present an architecture composed of a configurable array of computing cores and memory blocks in which both the execution mechanism and configuration principle of the computing cores and the behaviour of the memory blocks are based on streaming and dataflow principles. We illustrate our ideas with the implementation of a long finite impulse response (FIR) filter where memory tiles are used to store intermediate results.**

I. Motivation and Related Work

Streaming application are common in modern multimedia and wireless applications, like for example Video and Audio processing. In streaming applications, efficiency can drastically be increased if the underlying execution mechanism is based on dataflow principles, i.e. the system starts the execution as soon as the required input data is available, in contrast to conventional load/store mechanism commonly found in imperative approaches.

In embedded computing, coarse-grain reconfigurable architectures are an emerging paradigm for efficient implementations of streaming systems. Those architectures usually combine a general purpose processor (GPP) as host controller with an array of small, reconfigurable processing elements that are interconnected to form a larger, reconfigurable multicore architecture. Cores in those arrays are usually small and contain only the ALU and some local storage. Often bigger external memory is added to be able to store for example intermediate results or to provide look-up tables.

There have already been published a number of papers on coarse-grain reconfigurable architectures, an extended overview on reconfigurable architectures can be found in [1] and [2]. MorphoSys [3] is a hybrid of a host CPU and a reconfigurable array. The connection to external memory is provided via the system bus with DMA. The programming principle is based on imperative programming. ADRES [4] is a combination of a very long instruction word (VLIW) processor with a tight connection to a reconfigurable grid. The two parts are connected via a multi-port register file. Data access is performed via load/store operations. The programming is C-based. XPP [5] contains an array of 8x8 processing elements including 2 RAM-blocks per row. The XPP array can be programmed either with the low-level NML (native mapping language) or with an XPP-specific subset of C. Memory access is performed with read and write operations. DREAM [6] consists of a control unit, data path and a memory access unit. To transfer data between DREAM and the host CPU, exchange buffers are available. DREAM is programmed using macro-instructions that are described in single-assignment C syntax. RICA [7] is a heterogeneous array of reconfigurable ICs. Dedicated control ICs are available so that RICA does not require a host control CPU. In the array, distributed memory elements are available that are accessed via special memory access ICs. The programming principle is C-based.

All the presented architectures have in common that they rely on an imperative programming approach and, thus, on a read/write method to access their memories. Whenever a certain core requires data from an external memory, it requests the data. That request can be handled via a central control unit, a memory access unit or even a dedicated IC for memory access.

Here we present a data-driven approach, where other approaches are mostly control-driven. In our architecture, the complete execution scheme is stream-based. That means, input data is expected to be available as a stream, the elements in the architecture consume and produce streams, and a stream (or multiple streams) of data is the final result. In order to achieve such an architecture, each actor was designed based on stream-based execution. In contrast to the publications mentioned above, we consider the memories as streaming actors. Our basic assumption is that a core usually does not only need a single element of data from a certain memory but a stream of data elements. A common use case for this assumption is that a certain streaming application is partitioned into several parts that are executed in sequence. The intermediate results have then to be stored in a memory and streamed back into the system at a later stage. By implementing the memories as streaming actors, the data transfer has only to be initialised once. After the initialisation, the data then automatically streams to its destination. As a consequence, dedicated load/store operations can be omitted.

In the remainder of this paper we will illustrate the proposed architecture, focusing on the streaming memory actor. Finally we will present an extended use case where different features of the architecture are exploited.

II. Architecture

Figure 1 shows our architecture. The blocks denoted with **Cxy** represent simple reconfigurable cores with identifier **xy**, memory tiles are represented by the blocks labeled **My** where **y** is the identifier.

978-1-4673-2896-8/12/$31.00

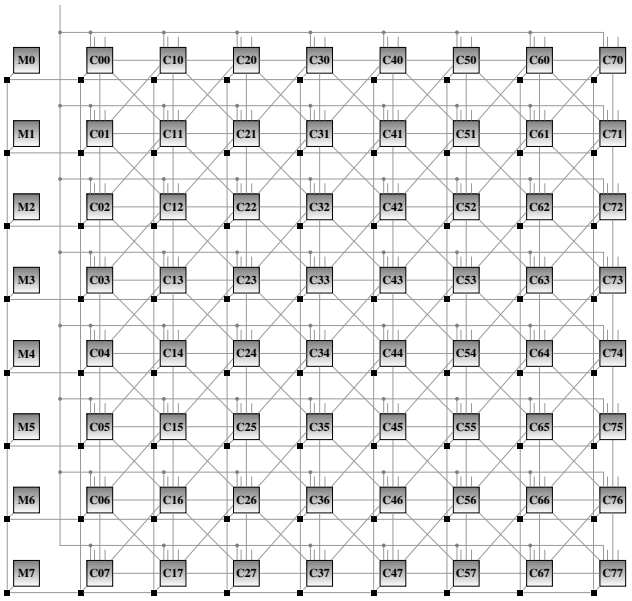
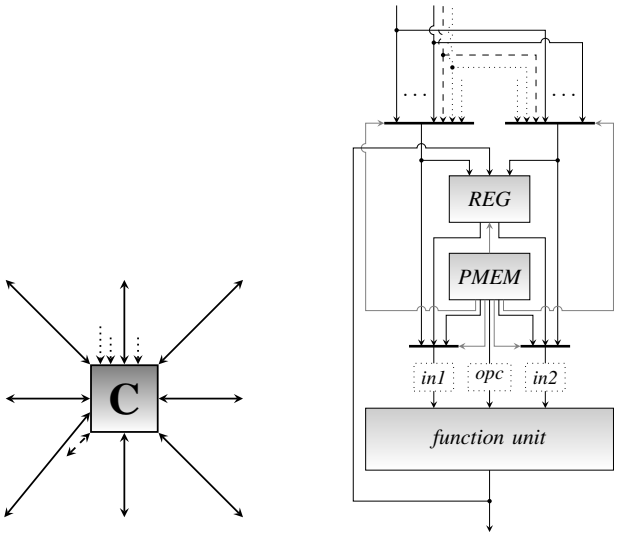


Figure 1: Architecture



(a) Connection of one core (b) Details of one core

Figure 2: Core

A. Cores

In Figure 2 two different views on a single core are shown. Figure 2a shows the connectivity and Figure 2b illustrates the internal composition. A core consists of:

1) Inputs: Figure 2a shows a close-up of one core of the grid shown in Figure 1 with a focus on the connectivity. The connectivity can hereby be split into three kinds: the local connection to neighbouring cores (the continuous lines), the global connection to the system via the NoC (the dashed line), and the external inputs (the dotted lines).

In Figure 2b, the same scheme is used. At the input of the core the incoming signals are connected to two multiplexers from which the correct input is selected according to the current setting in the program memory. Each multiplexer also includes one FIFO buffer to store incoming data (not shown in the graph).

2) Function unit: The function unit is responsible for the actual computations. It supports binary operations, such as numerical operations (e.g. addition, multiplication..), shifting operations and operations on the bit-level (e.g. and, or ..). Both integer and fixed-point operations are supported.

3) Local Memory: A local storage is available in form of a register file, denoted *REG* in the figure. It has three write ports and two read ports, the size of the register file can be parametrised during design-time.

4) Program Memory and Control: The program memory, labelled *PMEM*, is responsible for the actual control of the operation, the selection of the correct input, and the storage of data. It is configured with a finite-state-machine based principle which will be explained in the following section.

5) Configuration: The architecture is configured on two levels. One level represents the local view on a single core, i.e. the behaviour of one core. The other level represent the global view on the complete architecture, i.e. the flow of data in the array.

The configuration of the local view is based on dataflow principles and finite state machine (FSM) methods. The behaviour is a sequence of FSM stages, where for each stage the number of repetitions is indicated in the upper left corner of the FSM states in Figure 3. Each stage is then defined in terms of a dataflow graph with tokens on the arcs representing that either a token is required (at the inputs) or a token is produced (on the output). Furthermore, for each token it is defined where it comes from (at the inputs, *EX* represents an external input, *Rx* represents a value stored at register *R0*) or where it has to be stored (at the output).

For illustration, we use the example of a pipelined multiply-accumulate (MAC) operation on data streams. The MAC operation on the streams x and y is defined as follows:

$$mac = \sum_{i=0}^{N} x_i y_i = x_0 y_0 + x_1 y_1 + x_2 y_2 + \ldots + x_N y_N \quad (1)$$

For illustration purposes the mac operation is implemented in a pipelined fashion using separate stages for the multiplication and addition. The implementation of the complete mac operation requires three stages, of which the first one is an initial stage. In Figure 3, the configuration is shown, in Figure 4, the corresponding execution on the core is shown.

The first stage is labelled *S0*, which corresponds to Figure 4a. Here, the two external inputs (x_0 and y_0 from Equation 1) are multiplied and stored in the register file at *R0*. Following, the stage *S1*, which corresponds to Figure 4b, is executed, which represents a multiplication of x_1 and y_1 . The result of this multiplication is stored in *R1*. The final stage *S2*, shown in Figure 4c, performs an addition on the results of the *S1* and *S2* and stores the result in the register file. From here on, the core alternates between the stages *S1* and *S2*.

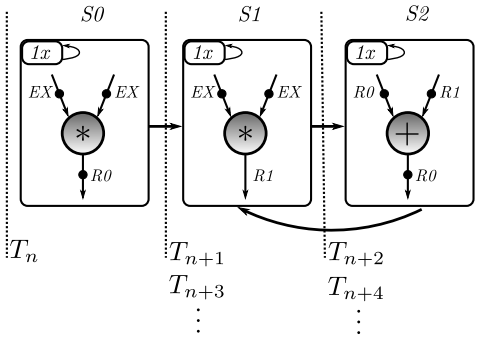
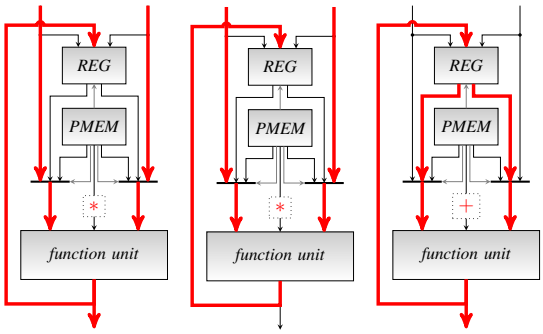


Figure 3: Configuration principle



(a) S0 (b) S1 (c) S2

Figure 4: Implementation of a MAC operation on one core

B. Interconnect

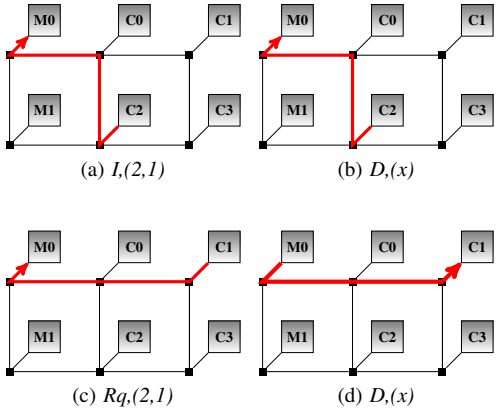
In the grid, three levels of interconnects are available: local nearest neighbour connections to support locality of reference that are implemented as point-to-point links between the nodes, a global Network on Chip (NoC) to enable full connectivity of the system without the need for a fully connected network, and a broadcast network to provide an input sample simultaneously to (a subset of) all cores in the grid.

C. Memory

Each incoming packet to the memory actor is one of the following types:

1) *data*: A packet identified with *D* followed by the actual data, sent by a producer
2) *index*: A packet with the type identifier *I* followed by the identifiers of the source and final destination of the data, sent by a producer
3) *request*: A packet with the type identifier *Rq* followed by the identifiers of the source and final destination of the data, sent by a consumer.

A complete cycle of data transfer is illustrated in Figure 5. In this transfer, the core **C2** sends data to the memory **M0** which is at a later stage requested by core **C1**. The cycle is as follows: in the beginning, **C2** sends an index packet containing its identifier and the destination's identifier, in this case *I,(2,1)*, to **M0**. This is shown in Figure 5a. Then, **C2** sends a stream of data to **M0**, shown in Figure 5b. At a certain point in time, **C1** sends a request packet to **M0** again with an identifier-tuple



(a) *I,(2,1)* (b) *D,(x)*

(c) *Rq,(2,1)* (d) *D,(x)*

Figure 5: Complete data transfer

consisting of the source of its data and its own identifier, in this case *Rq,(2,1)*. This is shown in Figure 5c. **M0** will now stream data to **C1**. This is shown in Figure 5d.

III. Use Case

The presented use case is a finite impulse response (FIR) filter [8], which is often used in the domain of digital signal processing, for example as high or low pass filter in digital audio processing.

The definition for a FIR filter is as follows:

$$y[n[= \sum_{k=0}^{M} b_k x[n-k] \\ = b_0 x[n] + b_1 x[n-1] + \ldots + b_M x[n-M] \quad (2)$$

where $y[n]$ is the current output sample, $x[n]$ the current input sample, b a list of filter coefficients and $M+1$ the length of the filter. A graphical representation is shown in Figure 6. The white rectangles represent unit delay elements, the circles represent the operations.

For long FIR filters it is often desired to partition the filter in parts which are executed in sequence. A general principle how this can be done is illustrated by the dotted rectangles in Figure 6. In this illustration, the FIR filter is partitioned into N parts of length four. The principle is quite simple: the results from one part are used as input for the next. In Figure 6, the results from the first part, denoted P_1 are streamed into the adder-row from the next part, denoted P_2. The same principle is applied to all the following parts. The final result is available at the output of P_N. If the parts are executed in sequence, the intermediate results from the parts should be stored in a memory. Consequently, the delay elements between the stages are replaced by a streaming memory actor.

A. Implementation on the proposed Architecture

On the 64 cores of our architecture, a 32-Tap FIR filter can directly be executed (each tap requires two cores, one for multiplication and one for addition). The mapping of a 32 tap filter can be seen in Figure 7. The highlighted connections

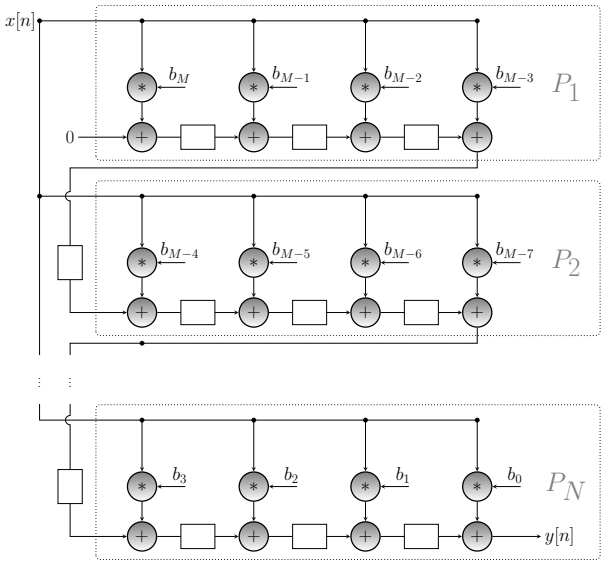


Figure 6: FIR filter - partitioned

correspond to the flow of data. Note hereby that the vast majority of the communication is implemented using the nearest neighbour network, as the FIR filter includes a high locality of reference. The delay elements are implemented by the FIFO buffers in the cores. The continuous lines are for one stage of a pipelined FIR filter, the dashed lines represent the communication between the different pipeline stages via a memory tile.

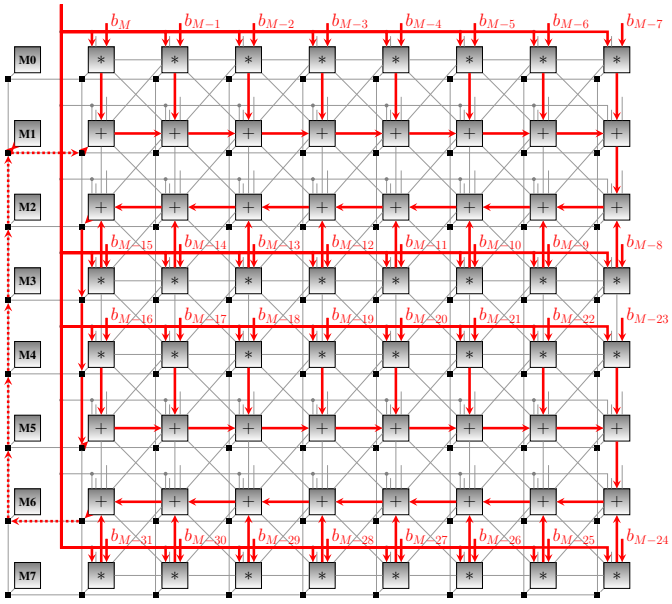


Figure 7: FIR filter

The coefficient b_M corresponds to b_{31}, coefficient b_{M-1} to b_{30} etc. The top left core (**C00** using the naming scheme of Figure 1) is the left-most multiplication node in Figure 6, i.e. it implements the multiplication of the input with coefficient b_{31}. Core **C10** implements the multiplication with b_{30} and so on. The filter taps *M-8* to *M-15* are handled by the rows 2 and 3 (with row 0 being the top row). The next set of filter taps, *M-16* to *M-23* are handled by the rows 4 and 5. Note hereby that the connection between core **C02** (the filter tap with *M-15* and core **C05** (the filter tap *M-16* is implemented via the NoC. The remaining filter coefficients are handled by the last two rows.

For the execution of a pipelined FIR filter, the principle shown in Figure 6 is used. Furthermore, the memory transfer protocol explained in Section II-C and shown in Figure 5 is used to store intermediate values between the FIR partitions on memory **M1**. As an example, we illustrate the execution of a 128-tap FIR filter partitioned into four parts P_1 to P_4. For each part, the mapping for one 32-tap FIR filter as used in Figure 7 is used. The coefficients are adapted to each stage. First, P_1 is executed using the coefficients b_{127} to b_{96}. The resulting data is sent to the memory **M1** via the NoC, as it is shown in Figure 7. Then, P_2 with coefficients b_{95} to b_{64} is executed. For this, the results from P_1 are streamed into the grid (to core **C8**, see Figure 7). Furthermore, the results from P_2 are streamed into **M1** via the NoC, just as in the previous stage. P_3 is similar to P_2, just that the coefficients b_{63} to b_{32} are used. In P_4, the final result is produced.

IV. Conclusion

A coarse-grain reconfigurable architecture targeted towards streaming applications was presented, in which all blocks, including memory blocks, are streaming actors. The actors are configured by a finite state machine (FSM) logic, the flow of data in the architecture is organised in a dataflow manner.

We showed that our configuration principle gives a straightforward implementation on the presented coarse-grain reconfigurable architecture of DSP applications, as illustrated by a partitioned FIR filter.

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.

[3] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, 2000.

[4] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field-Programmable Logic and Applications*. Springer, 2003, pp. 61–70.

[5] V. Baumgarte, G. Ehlers, F. May, A. Nückel, 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.

[6] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, P. Rolandi, C. Mucci, A. Lodi, A. Vitkovski, and L. Vanzolini, "A dynamically adaptive dsp for heterogeneous reconfigurable platforms," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. IEEE, 2007, pp. 1–6.

[7] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 75–85, 2008.

[8] J. McClellan, R. Schafer, and M. Yoder, *Signal processing first*. Pearson/Prentice Hall, 2003.