# Potential of an Interconnection Mode Dataflow-core

Lorenzo Verdoscia
Institute for High Performance Computing and Networking
CNR - Napoli, Italy
Email: lorenzo.verdoscia@na.icar.cnr.it

Roberto Giorgi
Dept. Ingegneria dell'Informazione
Università di Siena, Italy
Email: giorgi@dii.unisi.it

***Abstract*—In this paper we discuss a new type of core called the Dataflow-core. This core provides data processing based on data flow instructions rather than control flow instructions. Data flow instructions serve both to describe a program and to dynamically change the context of a dataflow program graph inside the accelerator of the Dataflow-core–on-the-fly. Our intended design aims to combine the performance of a fine-grained dataflow architecture with the flexibility of reconfiguration, without requiring a partial or new bit-stream for re-programming it, as is the case with FPGAs. The potential of the dataflow implementation of a function or functional program can be exploited simply by relying on its description through the dataflow instructions that re-program the core. Besides, inside the core both temporary data and instructions are eliminated as traffic over the memory.**

## I. INTRODUCTION

Spatial reconfigurable computing with FPGAs, massively-parallel systems based on soft-cores, and Coarse-Grained Reconfigurable Arrays cores accelerate applications by distributing operations across many parallel computational resources [1]. However, these cores have the disadvantages of reduced processor performance, higher power consumption and larger size [2] compared to a configurable dataflow core. While configurable computing has revealed its effectiveness over parallel systems based on conventional core processors [3], how to efficiently organize resources available at 14-nm technology or less in terms of programmability and low power consumption in more generally parallel architectures remains an open question [4].
Dataflow configurable architectures were proposed by Miller and Cocke [5]. Unlike in a von Neumann-based machine, this class of "configurables" used data flow as a method for configuring a computer to directly execute dataflow graphs – the Interconnection and Search Mode Configurables – where the natural and inherent parallelism of a program was exploited during its execution, forming, thus, the basic model for dataflow machines [6]. Even though several dataflow architectures [7] have been proposed, most of them fall into the search mode configurable [8]. Only one is partially of an interconnection mode type and partially of a search mode type [9], whereas our configurable Dataflow-core (hereinafter DF-core) proposal is of the interconnection mode architecture type. Here we discuss a new concept of core that eliminates both temporary data and instructions as traffic over the memory, that can be effectively and efficiently supported also by FPGAs, with adequate interconnect resources, or ASICs.

The idea is to make available a core that provides data processing after loading dataflow instructions, according to the demand-data driven co-design approach [10] rather than control flow instructions. In addition, dataflow instructions serve here both to describe a program and to change its structure on-the-fly, without the need for a partial or full reconfiguration. Our intended design aims to provide the performance of an interconnection mode dataflow architecture and the flexibility of reconfiguration without the need for passing a new bit-stream like in FPGAs.
The remainder of this paper is organized as follows. Section II presents the DF-core ISA; Section III describes the DF-core architecture, Section IV provides our conclusions.

TABLE I. DF-CORE OPERATOR SET

| | | | | | | |
|---|---|---|---|---|---|---|
| Arithmetic | ADD | SUB | MULT | | | |
| Comparison | EQ | NEQ | GE | GT | LE | LT |
| Special | ABS_ | LST | SL | SR | | |

## II. THE DF-CORE INSTRUCTION SET

In contrast to a conventional core processor that is mainly based on a RISC architecture, the DF-core has a custom architecture derived from the co-design process between the functional programming and the dataflow execution principles, given their strict relationship. The former creates a dataflow program graph by demanding a function for its operands (lazy evaluation) driven by the need for the function values. The latter executes such a graph in dataflow mode by consuming operands (eager evaluation). In our case we used an FP-based programming language [11] together with the *homogeneous* High Level Dataflow System (*h*HLDS) model [12].

### A. hHLDS overview

Briefly, in *h*HLDS a dataflow program graph (DPG) is a directed fine-grain graph where nodes are operators (actors) or links (places to hold tokens) with homogeneous I/O conditions on actors–they can only have exactly one output and two input arcs and can consume and produce only data tokens–while links represent only connections between arcs rather than pointers to other dataflow instructions. To preserve the I/O condition homogeneity, merge, switch and logic-gate actors are not present, therefore actors cannot produce control tokens. Links can be: i) Joint, where two or more input arcs can coexist; ii) Replica, where the output is reproduced over more than one arc. While actors are determinate, joint links are not determinate. Despite the model simplicity, with the *h*HLDS it is always possible to obtain determinate DPGs including data-dependent ones (proofs are given in [12]). Furthermore, the DPG evaluation is strict [13] but actors only fire when two input tokens are valid[1]. So no feedback interpretation or synchronization mechanism is needed to execute a DPG correctly, and its execution is completely asynchronous.

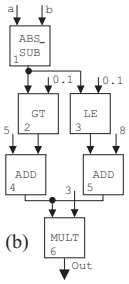
### B. The elemental operator set

The operator set we obtained in co-design, shown in Table I, is both functionally complete – more complex func-

[1]The validity is the intrinsic token's property that allows an actor to fire.

| $\alpha_i$ | | L | R | O |
|---|---|---|---|---|
| 1 | ABS_SUB | %% | %% | 2-3 |
| 2 | GT | 1 | 0.1% | 4 |
| 3 | LE | 1 | 0.1% | 5 |
| 4 | ADD | 5% | 2 | 6 |
| 5 | ADD | 3 | 8% | 6 |
| 6 | MULT | 4-5 | 3% | out |

(a)




(b)

Fig. 1. Example of a program DPG: (a) the DF-core assembly language, (b) the graphical representation.

tions (higher order) are created combining such operators by means of the metacomposition rule [11] – and consistent with *h*HLDS. When compiled with a program written in FP style that includes such operators, the produced DPG, or a context[2] of it, can be directly executed by the DF-core. As the set does not include logical operators (and, or, xor) because *h*HLDS does not support control tokens, their functionality can be expressed by higher order functions. For the division operation, when the divisor is a constant value, its reciprocal is computed at compile time, but a general solution is work in progress. The special operator ABS_, prefixed to an arithmetic operator, produces the absolute value of the corresponding operation. LST is the loop start operator employed in data-dependent cycles [14], and SL and SR are the operators that select the left or right actor token respectively.

### C. The assembly language

The DF-core provides programming in a custom assembly language which is the same graphical representation language that describes the dataflow graph of a program. A program in this assembly language is a collection of standard instructions named *expressions* that form a DPG. Each expression refers to an actor and specifies its functionality:

| $\alpha_i$ | $\Omega$ | $\tau_L$ | $\tau_R$ | $\delta_O$ |
|---|---|---|---|---|

where $\alpha_i$ represents the actor identifier, $\Omega$ is the operation the actor performs, $\tau_L$ and $\tau_R$ are the left and right input tokens, and $\delta_O$ is the actor or actor list of identifiers separated by the **-** (dash) character that has/have to receive its operation result. If the result is final, $\delta_O$ is tagged *out*. Regarding $\tau_L$ and $\tau_R$, the language distinguishes external and internal data values.
*External data*: if the data is known at compile time, its value starts with the **%** character. Once consumed, it becomes invalid, i.e., unable to fire an actor. If it is known at run time (e.g., produced by an external event), its value is represented with two **%** characters. If it is a constant value in the program, its value ends with the **%** and remains valid until the context changes.
*Internal data*: This is the value produced by a previous actor for another actor. It remains valid until the producer actor fires again. It is an integer that represents the numeric identifier of the actor that produced such data.

As an example, consider a sample program that receives in streaming couples of `a` and `b` values in order to compute their absolute value. If the value is greater than `0.1`, the token `5` is selected, otherwise the token `8`. Finally, the result is scaled by a factor of `3`. The code and its graphical representation are shown in Fig. 1.(a) and Fig. 1.(b) respectively. We would like to point out that the two actors GT (greater) and LE (less or equal) are mutually exclusive, so only an actor that satisfies the predicate produces a valid token whose value is 0, while the other produces a non-valid (don't care) token. This feature simplifies the design of the DF-core accelerator making it possible to use only identical Dataflow Functional Units (DFUs) and only data wires to connect them.
During the translation of an assembly code, the assembler generates three machine codes that describe a DPG operation: the *graph interconnect code*, which defines the interconnection between actors; the *actor operating code*, which defines, for each actor, its operation and role in the DPG; and the *input token code*, which defines the tokens assigned to initial actors to start the computation. Unlike with conventional instructions, this split makes it feasible to run a DPG by first configuring the accelerator within the DPG context and then activating its execution via the program input tokens. It is possible to overlap an execution and a new context preloading.

[2]With the term context we call a DPG that fits in the accelerator. Larger DPGs are partitioned in suitable contexts.

## III. The Dataflow-core Architecture

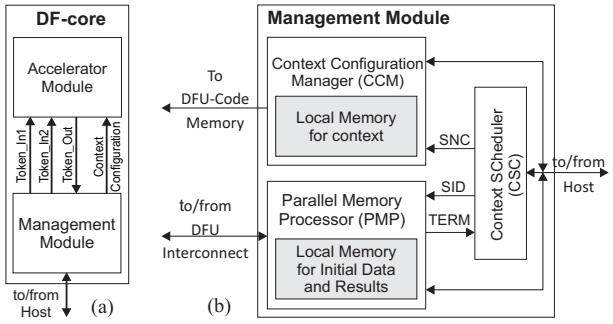



Fig. 2. (a) the Dataflow-core, (b) the Management Module.

Several reasons shaped the design of the Dataflow-core (DF-core). First, we wanted to map dataflow graphs onto hardware in a more flexible way than traditional HLS tools allow. Second, we wanted to combine straightforward data flow control with an actor firing mechanism at a minimal hardware cost. Third, we wanted to avoid the traffic generated by load and store instructions in order to improve performance. Finally, we wanted to explore the possibility of using primitive functions of a functional language for a more effective translation into data-flow assembly.
The DF-core consists of two main modules (Fig. 2.(a)):

- Management Module, dedicated to managing the DPG contexts and/or the data tokens list for execution on the Accelerator Module;
- Accelerator Module, dedicated to executing DPG contexts.

### A. The Management Module Architecture

It is constituted by three fundamental submodules (Fig. 2.(b)):

- *Context Configuration Manager* (CCM). Once the contexts generated by the compiler are stored in the Context Configuration Memory (a small local memory), they can be loaded dynamically into the accelerator as soon as the SNC signal (Send-Next-Configuration) is activated by the Context Scheduler submodule.

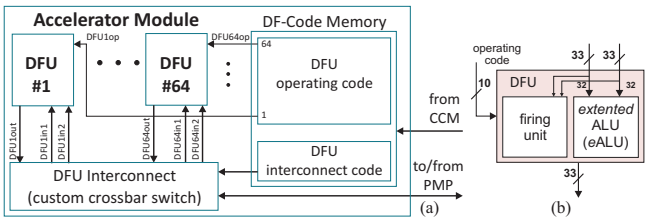



Fig. 3. (a) Accelerator Module, (b) DFU.

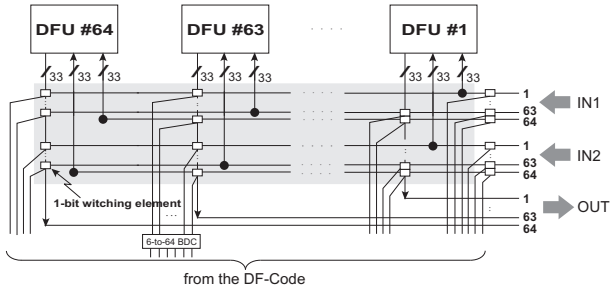



Fig. 4. DFU interconnection network.

- *Parallel Memory Processor* (PMP). It takes care of the initial input data and collects the final output data that are processed by the accelerator. Once the scheduler enables the SID (Send-Initial-Data) signal, the PMP: (i) Prepares the transfer of the initial data tokens to the accelerator module; (ii) Collects the result data tokens as soon as they are ready at the output buffer registers; (iii) Sends a termination signal to the scheduler when the computation ends.
- *Context SCheduler* (CSC): (i) Implements the scheduling policy (defined after the partitioning and mapping activities) for the contexts allocated on the CCM; (ii) Sends enabling signals to the CCM and to the PMP; (iii) Manages the interaction with the Host.

### B. The Accelerator Module Architecture

The accelerator, shown in Fig. 3.(a), is composed of a DF-Code Memory, a custom crossbar switch (DFU Interconnect) and $k = 64$ identical and thin DFUs[3] (a possible instance).

***DF-Code Memory***. It stores the contexts ready for execution. The *DFU interconnect code* memory is a register that is dedicated to storing the interconnect code ($2 \times k \times \lceil \log_2 k \rceil$ bits). The *DFU operating code* memory is a register collection ($k \times 10$-bits) that stores the operating code for each DFU. To simplify the transfer of information from the Management Module to the accelerator there is a dedicated bus under the supervision of the Management module.

***Dataflow Functional Unit***. A DFU (Fig. 3.(b)) implements any *h*HLDS actor. It consumes two 33-bit (32-bit data and 1-bit validity) valid tokens (DFUIn1 and DFUIn2) and produces one 33-bit token (DFUOut). If the token is invalid, its validity bit is set to 0. A DFU is composed of an *extended* ALU (*e*ALU) (arithmetic, multiplier, and etc.) that implements the operator set, a firing unit that ensures the right behavior, and of a 10-bit *operating code* register, which holds 5-bit for the *e*ALU operations and 5-bit for the firing unit that specify the DFU context (constant token, token streaming, loop, pipelining, and conditional participation).

[3]The actual number of DFUs, the associated DFU Interconnect, and the DF-Code memory can vary according to the available on-chip resources.

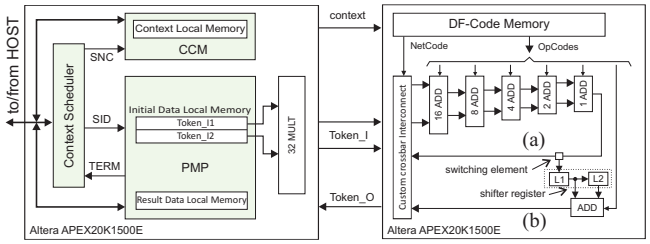



Fig. 5. DF-core organization for the inner products. Device-2: (a) $n = 32$, (b) $n = 64$.

***DFU Interconnect***. It consists of a custom crossbar grid of wires connected by switching elements (Fig. 4) that allow for the connection of any DFU output to any DFU input, except itself, or to the PMP in the Management Module. All switches along a column are controlled by a $\lceil \log_2 k \rceil$-to-$k$ decoder whose control-signals come directly from the dedicated registers of the DFU Interconnect code memory. Across a row only a valid token can exist.This feature is essential for implementing conditional and cyclic structures in conformance with the *h*HLDS model.
Because the interconnect handles a large number of inputs and outputs, it is a crucial component of this architecture. Therefore, its sizing is chosen based on chip capabilities. Nevertheless, it is possible to implement a crossbar interconnecting 128 tiles with an area cost of 6% of the total [15].

### C. DF-core characterization

Our DF-core instance consists of $k = 64$ DFU units and executes operations on 32-fixed-point operands. For its implementation we used a custom board demonstrator with two Altera APEX20K1500E devices. Device-1 is dedicated to the implementation of the Context module plus 32 DFUs. Device-2 is dedicated to the implementation of the Accelerator module with 32 DFUs and the custom crossbar Interconnect (due to the Interconnect area penalty). Please note that in a previous paper [14] we evaluated the latencies of a DFU, register-to-register, and the context switch which are 30 ns, 7 ns (device-to-device registers) and 4 ns (internal registers), and 32 ns respectively.

### D. Potentiality of the DF-core: an evaluation test case

The test case evaluates the product of matrices $C(n,n) = A(n,n) \times B(n,n)$ [16]. In the dataflow assembly it consists of $n^2$ independent inner products (IPs) whose DPGs are organized in identically reversed binary-trees, each formed of $n$ multiplications and $n-1$ additions sequenced in $\lceil log_2 n \rceil + 1$ levels. By virtue of its DPG shape, the inner product is well

suited for a naturally pipelined execution. Anyway, thanks to the DF-core architecture and the assembly language, pipelined execution can be used for any DPG computation if initial data occurs in streaming manner. Here we do not discuss this because out of our purpose.
For the test we used $n = 32$ and $n = 64$, with all matrix elements reside in the local data memory of the Context Management module. For $n = 32$ the inner product (IP) DPG is wholly mapped onto the DF-core, by means of 63 of the 64 DFUs available (Fig. 5.(a)). For $n = 64$ we first split the DPG into two sub-DPGs as in $n = 32$, then we execute the two inner products IP'$= n_{a'_{i/2}} \times n_{b'_{j/2}}$ and IP"$= n_{a''_{i/2}} \times n_{b''_{j/2}}$, and use the DFU #64 to add the two results as shown in Fig. 5.(b). We would like to point out that this decomposition does not hinder the throughput of the pipeline because the DPG in Fig. 5.(b) behaves as if all the required DFUs were in the DF-core but the number of IPs to execute doubles.

*1) Matrix multiplication execution:* The parallel execution of the 32 multiplications on the Management module requires the transfer of $2 \times 32 = 64$ tokens (In1 and In2) from the PMP local memory to the 32 DFUs. As a token is 33 bits, the total number of bits to transfer to the 32 DFUs is $33 \times 64 = 2112$. By using the internal FPGA interconnect, we transfer $2 \times 4$ tokens ($264 + 264$ bits) at a time. In total this takes $8\times$ internal register-to-register transfers and has a latency equal to $8 \times 4 = 32$ ns, while the multiplication needs 30 ns. Transferring $32 \times 33$-bit tokens of the product from the Management module to the accelerator requires $4\times$ external register-to-register transfers and latency of $4 \times 7 = 28$ ns, whereas inside the accelerator the transfer from the input buffers to the first 16 DFUs requires latency of $4 \times 4 = 16$ ns.

When $n = 32$, we need 30 ns for each additional level ($log_2(32) + 1 = 6$ total levels) and 11 ns to transfer $c_{ij}$ back to the management module. Therefore, the pipeline requires a number of stages $n_s = 11$ to fill the pipeline with a stage latency $\tau_s = 32$ ns (clock rate 31,2MHz) with $\tau_s = Max(\tau_{s_i})$. The total number of cycles $n_c$ required for the matrix multiplication is $n_c = 32^2 + 11 = 1035$.

When $n = 64$, the DFSC processor requires the same latencies as with $n = 32$ up to the DFU #63. Then the DFU #64, through the two cascaded latches L1 and L2 in Fig. 5.b, produces $c_{ij}$. In this case we add the first latch latency (4ns) to the DFU #63 latency while the second latch latency is added to the DFU #64 latency. Consequently, we have $\tau_s = 34$ ns, $n_l = log_2(64) + 1 = 7$ number of levels in the accelerator, and $n_s = 13$, while the number of IPs doubles. The total number of cycles $n_c$ required for the matrix multiplication, in this case, is $n_c = 2 \times 64^2 + 13 = 8205$.
As we can see, the results show that a DF-core really flushes out traffic over the memory during a computation and reduces its complexity to $O(n^2)$ in comparison with a conventional core where the computation complexity of the matrix multiplication algorithm is $O(n^3)$ included the number of memory operations.

## IV. Conclusions

This paper presented a new type of core, named Dataflow-core (DF-core), which executes dataflow program graphs by means of a dataflow computing module commanded by an ad hoc memory module, instead of control flow instructions. As well as dataflow architectures proposed till now, which eliminate the program counter, this new concept of core eliminates both temporary data and instructions as traffic over the memory, as well. This feature is obtained through a configurable interconnection that connects identical and thin dataflow functional units. Moreover, such units are able to execute any operator of a set of them originated in co-design between the homogeneous High Level Dataflow System and a programming language in Backus-style FP. We believe that this concept needs further development but represents a first step toward a more flexible execution of generic programs on a scalable, reconfigurable platform. The DF-core introduces a new level of programmability that enhances the utility of FPGA platforms through the use of dataflow program graphs rather than pretending to convert Control-Flow instructions.

## References


[1] A. Chattopadhyay, "Ingredients of adaptability: A survey of reconfigurable processors," *VLSI Des.*, vol. 2013, pp. 10:10–10:10, Jan. 2013. [Online]. Available: http://dx.doi.org/10.1155/2013/683615

[2] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, "Application-specific customization of parameterized fpga soft-core processors," in *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, Nov 2006, pp. 261–268.

[3] M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec, and M. Valero, "Moving from petaflops to petadata," *Commun. ACM*, vol. 56, no. 5, pp. 39–42, May 2013.

[4] S. H. Fuller and L. I. Millett, "Computing performance: Game over or next level?" *Computer*, vol. 44, no. 1, pp. 31–38, 2011.

[5] R. Miller and J. Cocke, "Configurable computers: A new class of general purpose machines," in *International Symposium on Theoretical Programming*, ser. Lecture Notes in Computer Science, A. Ershov and V. A. Nepomniaschy, Eds. Springer Berlin / Heidelberg, 1974, vol. 5, pp. 285–298.

[6] J. Chudík, *Algorithms, software and hardware of parallel computers*. London, UK, UK: Springer-Verlag, 1984, ch. Data flow computer architecture, pp. 323–358.

[7] M. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi, *Guide to DataFlow Supercomputing*. Berlin, DE: Springer, Apr 2015.

[8] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-neumann architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1489–1509, June 2014.

[9] K. Gostelow and Arvind, *A Computer Capable of Exchanging Processing Elements for Time*, ser. Technical report. Department of Information and Computer Science, University of California, 1976.

[10] L. Verdoscia and R. Vaccaro, "Position paper: Validity of the static dataflow approach for exascale computing challenges," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2013*, Edinburgh, Scotland, UK, Sep. 8, 2013, pp. 38–41.

[11] J. Backus, "Can programming be liberated from von Neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, pp. 613–641, Aug. 1978.

[12] L. Verdoscia and R. Vaccaro, "A high-level dataflow system," *Computing*, vol. 60, no. 4, pp. 285–305, 1998.

[13] J. B. Dennis, "Data flow computation," in *Proc. of the NATO Advanced Study Institute on Control flow and data flow: concepts of distributed programming*. New York, NY, USA: Springer-Verlag New York, Inc., 1986, pp. 345–398. [Online]. Available: http://portal.acm.org/citation.cfm?id=22086.22093

[14] L. Verdoscia, R. Vaccaro, and R. Giorgi, "A clockless computing system based on the static dataflow paradigm," in *Proc. IEEE Int.l Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM-2014)*, aug 2014, pp. 1–8.

[15] G. Passas, M. Katevenis, and D. Pnevmatikatos, "A 128 x 128 x 24gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6% of their area," in *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, May 2010, pp. 87–95.

[16] L. Verdoscia, R. Vaccaro, and R. Giorgi, "A matrix multiplier case study for an evaluation of a configurable dataflow-machine," in *ACM CF'15 - LP-EMS*, May 2015, pp. 1–6.