A Clockless Computing System based on the Static Dataflow Paradigm

Lorenzo Verdoscia and Roberto Vaccaro
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

***Abstract*—The ambitious challenges posed by next exascale computing systems may require a critical re-examination of both architecture design and consolidated wisdom in terms of programming style and execution model, because such systems are expected to be constituted by thousands of processors with thousands of cores per chip. But how to build exascale architectures remains an open question. This paper presents a novel computing system based on a configurable architecture and a static dataflow execution model. We assume that the basic computational unit is constituted by a dataflow graph. Each processing node is constituted by an ad hoc kernel processor – designed to manage and schedule dataflow graphs, and a manycore dataflow execution engine – designed to execute such dataflow graphs. The main components of the dataflow execution engine are the Dataflow Actor Cores (DACs), which are small, identical and configurable.**

The major contributions of this paper are: i) the introduction of a machine language (named D#) which represents the low-level static configuration information of the system; ii) the introduction of a self-scheduled clockless mechanism to start operations on the presence of validity tokens only; iii) a design that avoids the need of temporary storage for tokens on the links of the DACs. Our preliminary tests on FPGA-based hardware show the feasibility of this approach.

I. Introduction

The next big HPC supercomputing challenge is to break the exascale barrier with processor chips expected to be constituted by thousands of cores [1] at 14-nm technology or less. New architectures may not be easily programmed. Our proposal is to organize the computational units so that in the same chip we have both standard cores and dataflow execution cores. If many-core is destined to be the way forward, a profoundly new architectural model must be developed. But their effective organization into a programmable architecture constitutes a list of challenging points such as the rethink of programming models so that programmers can express application parallelism. To successfully meet this impressive set of challenges, it is required a critical re-examination of conventional wisdom, in terms of programming style and execution model, and more bravery, to search for different solutions still unexplored. Moreover, there is growing agreement that reaching this goal will require a substantial shift toward hardware/software co-design [2][3]. From a programming and architectural point of view, such an approach, combining the functional programming style and the static dataflow paradigm, would give a significant contribution to the design of new and unconventional exascale computing systems, given their close relationship and natural ability to explore and deal with parallelism: the first can create dataflow program graphs in demand mode, the second can execute them in dataflow mode [4][5].

Even though dataflow paradigm is not new, recently it has received a renewed interest in Industries and research [6][7][8][9][10][11][12]. We show in this work that with our configurable[1] computing system it is possible to realize an asynchronous and configurable dataflow execution engine where the execution of a Dataflow Program Graph (hereafter DPG) happens directly in hardware exactly like a dataflow graph computation takes place in the *homogeneous* High Level Dataflow model [14]. We create the one-to-one correspondence between actors of the model and Dataflow Actor Cores (hereafter DACs) of the Dataflow Execution Engine using the demand-data driven approach as co-design methodology [15].

We point out that our configurable dataflow execution engine is different from existing FPGA-based dataflow graph processors because we do not need any kind of control flow graph mechanisms during the execution of a dataflow graph except the natural flow of data even if the graph includes cycles.

This paper makes three main contributions:

(1) we introduce a machine language (named D#) which represents the low-level static configuration information of the system or in other terms the Dataflow Program Graph; such language also represent a complete set of the elemental operator to support the Chiara functional language [16] and in accordance with the previously introduced model for the behavior of dataflow graph called *h*HDLS [14];

(2) the introduction of a self-scheduled clockless mechanism to start DAC operations on the presence of validity tokens only; differently from Dennis' *Dataflow Schemas* [17], where operations are triggered by the token presence, in our case we start operations on a /em token validity information; in such case Dennis' *Instruction Scheduler* and the check of token absence on any output arc of a DAC is eliminated - so, a DAC firing is self-scheduled;

(3) a design that avoids the need of temporary storage for tokens on the links of DACs: in our machine the DPG *arcs* are mapped to simple wires since they represent simple transmission of tokens rather than dependencies among operations; *links* are mapped to junction points rather than places to hold tokens. Therefore, in our case, tokens can flow asynchronously from a DAC to another, eliminating the need of any temporary storage to execute a DPG.

[1] As terms like *configurable* or *reconfigurable* computing [13] in the past had a more general meaning than nowadays it has, e.g. FPGA-based reconfigurable computing, to avoid confusion, from here on the newest meaning will be explicitly expressed by using the term FPGA-based.

The remainder of this paper is organized as follows. Section II briefly prefaces the link between configurable and dataflow architectures; Section III introduces D# language and shows as it is linked to the *homogeneous* High Level Dataflow System, the functional language Chiara, and the compiling tools; Section IV presents the configurable dataflow machine and its program execution model; Section V presents some useful parameters to compare a many-core dataflow architecture to others; Section VI discuss related work in this area; Section VII for our conclusions.

II. Configurable and Dataflow architectures

Data-flow languages and architectures derive from a base paradigm of the data-flow graph – that is, the modeling of a program as a set of operator and link nodes [17] interconnected by a set of data- and control-carrying arcs. Under this paradigm there is no current operation, and each operator is free to execute when its data arrives making explode, thus, parallelism at an inherently fine-grain level. But due to technological limits and the unsuitability of the DPG representation model for for their direct execution onto a realistic hardware dataflow execution engine [15], dataflow has received a harsh criticism. As nowadays most of the obstacles have been overcome, a new scenario opens up for dataflow. To better comprehend the reason why our architecture is configurable and our dataflow paradigm is the *homogeneous* High Level Dataflow System model, we report a brief retrospect of some pioneering works that originated the configurable computing.

In the 1966 Miller and Rutledge [18] presented an approach to automatically convert a sequential program into a parallel program whose technique produced a block diagram for parallel operations that depended on the data flow rather than instruction sequences. In contrast to a von Neumann-based machine, a few years later, Miller and Cocke [19] proposed a new class of configurable and general purpose computers that used this data-flow as a method for configuring such a computer to directly execute it – the Interconnection Mode Configurables and Search Mode Configurables, where the natural and inherent parallelism of a program was exploited during its execution. As no additional sequencing limitations due to the program instructions or to the machine control were imposed, this made the main difference in respect to the conventional way to sequence the computer operations. Their peculiarity was the possibility of dynamically interconnecting the processors that correspond the the actors in the dataflow program according to the graph.

The interconnection and search mode configurables can be considered as basic models of dataflow machines [20]. While in the former the interconnection of operational units is actually implemented through a configurable switching, in the latter the interconnection of operational units is simulated. However, even though a search mode architecture eliminated the program counter and the global updateable storage, it preserved the concept of enabled/executable instruction – on the other hand the instruction cycle inefficiency has been identified as one of the main shortcomings of earlier experience in dataflow computing. Whereas the interconnection mode architecture allowed the elimination of the enabled instruction concept also. In fact, an interconnection network is used to create direct interconnections between outputs of operational units and inputs to other operational units in a manner consistent with the data flow model structure. No instruction needed to be fetched from memory during the execution of an algorithm, and temporary data and instruction were eliminated as well as traffic over the memory access ports. In contrast to how the execution of DPGs should happen on an interconnection mode architecture, the actors and links as well as defined with the classical dataflow model [17] practically constitute an insuperable obstacle because (1) actors have heterogeneous number of I/O link and consume and produce heterogeneous tokens, data and control; (2) links are heterogeneous to hold tokens, data and control. So, all realized dataflow machines have fallen within the type search mode. Only the Arvind and Gostelow computer [21] is partially of interconnection mode type and partially of search mode type. In contrast, our configurable dataflow machine (sec. IV) falls into the interconnection mode architectures.

III. *h*HLDS, D# machine language, and compiling tools

A. The hHLDS model

High-Level Dataflow System (HLDS) [14] is a formal model to describe the behavior of a directed dataflow graph where nodes are operators (actors) or links (places to hold tokens) that can have heterogeneous I/O conditions. Nodes are connected by arcs along which tokens (data and control) may travel. In that paper, also the *homogeneous* HLDS (*h*HLDS) was presented. *h*HLDS describes the behavior of a static dataflow graph imposing homogeneous I/O conditions on actors but not on links. Actors can only have exactly one output and two input arcs and consume and produce only data tokens, links represent only connections between arcs. Since *h*HLDS' actors cannot produce control tokens, merge, switch, and logic-gate actors [17] are not present. While actors are determinate, links[2] may be not determinate. In contrast, these features simplify the design of a dataflow execution engine chip using only identical DACs and one type of connection among them. In addition, despite the model simplicity, in *h*HLDS it has been proved that it is always possible to obtain DPGs which are determinate and where:

- actors fire when their two input tokens are valid, i.e. able to fire an actor, and no matter if their previous output token has not been consumed. In this case, the new token shall replace the previous one. In a system that allows the flow of only data tokens, this property is essential to construct determinate cycles (loops);
- to execute a program correctly, only one way token flow is present as no feedback interpretation is needed;
- no synchronization mechanism needs to control the token flow, thus the model is completely asynchronous.

In *h*HLDS actors and links are connected to form a more complex DFG. However, the resulting DFG may be not determinate

[2]In *h*HDLS there exist two types of links: i) Joint links, which represent a place where two or more output arcs can coexist and ii) Replica links, which are similar to joint links but have only one output arcs. In the case of Joint links the output arc (among the several available) where the token will travel is unpredictable.

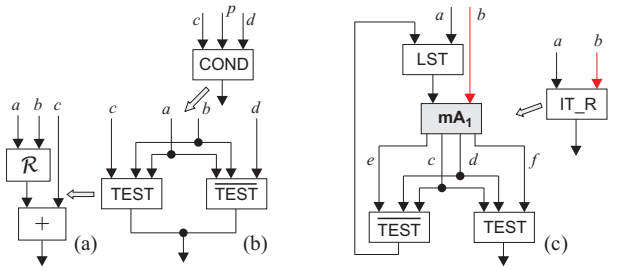


Fig. 1: The basic macro-Actors (mAs) in D#: (a) TEST, (b) COND, (c) IT_R

if cycles occur because no closure property can be guaranteed [22]. This happens for sure when the graph includes joint links, which are not-determinate. In the case when the DPG results to be determinate, we name it macro-Actor (mA). Obviously, an mA is characterized by having I(mA)$>$ 2 and O(mA)$\geq$ 1 where I(mA) is the number of input arcs (in-set) of mA and O(mA) is the number of output arcs (out-set).

B. The D# machine language

Our machine offers programming in a machine language that we call D#. D# is both the machine language and the graphical representation language that describes the dataflow graph of a program. It has been defined applying the demand-data-driven approach to co-design methodology [15] between the functional paradigm and the *h*HLDS paradigm.

When a DPG, i.e. the abstract entity in *h*HLDS, is embodied in the machine hardware, it happens that: (1) each DPG actor (abstract entity) is turned into one DAC, i.e. the physical entity, the actor firing rules become the DAC activation rules; (2) each arc/link (abstract entities) that connects two/more-than-two DPG actors is turned into a wire/wire-junction inside the interconnection network (physical entity) that connects two/more-than-two DACs; (3) each token and its validity (abstract entity) are turned into a data value and its validity signal (physical entity) so that the self-scheduling of a DACs can happen.

As macro-actors structures in D# are formed like in *h*HLDS, here we only report the fundamental ones that allow the creation of more complex structures (i.e., TEST, COND, IT_R macro-Actors).

The macro-Actor TEST. The simplest conditional structure is the mA TEST. It is an example of data-dependent DPG. When coupled to its complement $\overline{\text{TEST}}$, it forms a fundamental building-block to create conditional and iterative mAs. TEST is represented by a determinate and well-behaved mA with in-set = 3 and out-set =1 and formed connecting the relational actor $\mathcal{R}$ to the actor that performs the arithmetic operator **+** as shown in Fig 1(a). If $a, b, c \in \mathbb{R}$, its semantics is:

$$\mathsf{TEST}(a,b,c) = \begin{cases} c & \text{if } a\ \mathcal{R}\ b \text{ is satisfied} \\ \perp & \text{otherwise} \end{cases}$$

$\perp$ stays for not valid value. When the actor $\mathcal{R}$ satisfies its relation on the tokens *a* and *b*, it produces a token that has the data-value 0 (zero) and the validity "valid", thus the operation produces the token c. When the relational actor $\mathcal{R}$ does not satisfy its relation, it produces a token that has the data-value don't-care and the validity "not valid".

The macro-Actor COND. The simplest relational structure is the mA COND, shown in Fig. 1(b). It forms the building-block to create more complex conditional structures. COND is represented by a determinate and well-behaved mA with in-set = 4 and out-set =1. It is formed connecting the two mAs TEST and $\overline{\text{TEST}}$ with a link Joint. If $a, b, c, d \in \mathbb{R}$ and $p = a\ \mathcal{R}\ b$. Its semantics is:

$$\mathsf{COND}(a,b,c,d) = \begin{cases} c & \text{if } a\ \mathcal{R}\ b \text{ is satisfied} \\ d & \text{otherwise} \end{cases}$$

The macro-Actor IT_R. The iterative data-depend structure is the mA IT_R. It constitutes the building- block to create more complex data-dependent iterative structures. It is represented by a determinate and well-behaved macronode with in-set = 2 and out-set =1. IT_R is formed connecting the two mAs TEST and $\overline{\text{TEST}}$, an arithmetic actor or a macro-Actor mA_1, and the actor **LST** as shown in Fig. 1(c). The LST semantics is: it selects the right token the first time which is fired, the left token otherwise. If $a, b, c, d, e, f \in \mathbb{R}$, its semantics is:

$$\mathsf{IT\_R}(a,b,mA) = \begin{cases} \mathsf{IT\_R} & \text{if } c\ \mathcal{R}\ d \text{ is satisfied} \\ f & \text{otherwise} \end{cases}$$

Observing Fig. 1(c), we point out that, if mA_1 (at the center of Fig. 1(c)) is itself an IT_R, the figure represents a determinate and well-behaved nested-data-dependent iterative structure.

D# definition. *D# programming system is a tuple* $(\mathbf{A}, \mathbf{T}, \mathbf{F})$ *where: A is the set of actor-number identifiers, T is a set of tokens and the* undefined *special one*$\perp$ *(called* bottom*) – generally used to denote errors; and F is a set of operators from tokens to tokens.*

In the D# language a program is a collection of standard expressions that form a DPG. Each expression refers an actor and specifies its functionality. It is so organized:

$$< \mathbf{a} >, < \mathbf{f} >, < \mathbf{t}_L >, < \mathbf{t}_R >, < \mathbf{d}_O >$$

where **a** is the identifier number of the actor, **f** is its operation to perform, $\mathbf{t}_L$ and $\mathbf{t}_R$ are its left and right input tokens, $\mathbf{d}_O$ is the identifier actor number/numbers that has/have to receive its operation result. If the result is a final one $\mathbf{d}_O$ is tagged *out*. Regarding to $\mathbf{t}_L$ and $\mathbf{t}_R$, in the language we distinguish external and internal data values. The former are values starting with the **%** character when they are the initial values – they become not valid once consumed, and values terminating with the **%** when they are the constant values – they continue to be valid. The latter are integers that represent the identifier number of the actor which will produce that value. As an example, let us consider the first expression of the D# code shown in Fig. 2(b). The actor LST will receive its left input value from the actor numbered 26 while its right input value is an initial one. Regarding to $\mathbf{d}_O$, if it is a list of integers separated by **-** (dash) characters, this means that each actor in the list will receive the value produced. All the identifier actor numbers present in D# notation play the basic role both to correctly and simply generate the code for the configurable network inside the dataflow execution engine and to allow using available software tools that can efficiently carry out the mapping phase.

C. CHIARA language and the compiling tools

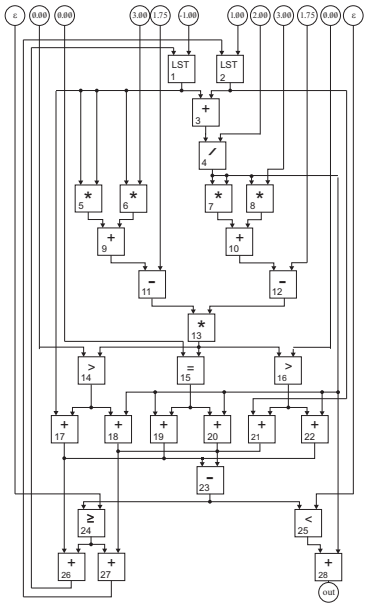
The CHIARA language has been already defined in [16], here is briefly recalled: a CHIARA program is nothing but a set of function definitions plus an expression (i.e. a function

```
# Fxlxm evaluates the Fx sign in the interval xl and xm
# After Fxlxm application, the new convergence interval can be
# xlxm = [xl, xm] or xmxr = [xm, xr] or xmxm = [xm, xm]
# if xm is the solution
# Pnt creates the sequence <xl, xm, xr> with xl, xm, and xr
# the left, average, and right interval values
# FxTest checks the convergence loop

def eps = 0.01%
def Fx = - o [ + o & * o [[ id ,id ], [ id, 3% ]], 1.75% ]
def Fxlxm = * o [ Fx o 1,Fx o 2 ]
def xm = / o [+ o [ 1, 2 ], 2% ]
def xlxm = [ 1, 2 ]
def xmxr = [ 2, 3 ]
def xmxm = [ 2, 2 ]
def Pnt = [1, xm, 2]
def Iter = case(lt o [Fxlxm ,0%]-->xlxm ;
          eq o [Fxlxm , 0%]-->xmxm ;
           lt o [Fxlxm , 0%]-->xmxr) o Pnt
def FxTest = lt o [- o [ 1, 2 ], eps]

#main program
1 o (repeat Iter, FxTest ): <-1, 1>
stop
```

(a) Chiara program code

Actor id	Oper	Left input	Right input	Output
1	LST	26	%-1	3-5-6-17
2	LST	27	%1	3-21
3	ADD	1	2	4
4	DIV	3	2%	7-8-18-19-20-22-28
5	MUL	1	1	9
6	MUL	1	3%	9
7	MUL	4	4	10
8	MUL	4	3%	10
9	ADD	5	6	11
10	ADD	7	8	12
11	SUB	9	1.75%	13
12	SUB	10	1.75%	13
13	MUL	11	12	14-15-16
14	LT	13	0%	17-18
15	EQ	13	0%	19-20
16	GT	13	0%	21-22
17	ADD	1	14	23-26
18	ADD	14	4	23-27
19	ADD	4	15	23-26
20	ADD	15	4	23-27
21	ADD	2	16	23-26
22	ADD	16	1	23-27
23	SUB	17-19-21	18-20-22	24-25
24	GEQ	23	0.01%	26-27
25	LT	23	0.01%	28
26	ADD	17-19-21	24	1
27	ADD	27	18-20-22	2
28	ADD	25	4	out

(b) D# language code



(c) D# graphical representation

Fig. 2: Zeroes of the function $f(x) = x^2 + 3x - 1.75$ with the bisection method: interval $[-1, 1]$ and approximation $\varepsilon = 0.01$

applied to an object) that, once evaluated, will represent the result of the program. The writing of complex programs is not a trivial activity a fortiori with both D# machine and graphical representation languages that do not provide higher-order function. To make more straightforward and exploit the powerful program that the functional style has in exploding all parallelism that programs present, we created the functional programming language CHIARA to program the machine.

CHIARA system is a tuple ($\boldsymbol{O}$, $\boldsymbol{F}$, $\mathcal{F}$ **:**, $\boldsymbol{D}$), where: **O** is a set of objects; **F** is a set of functions (or operators) from objects to objects; $\mathcal{F}$ is a set of functional forms (*functionals*) from functions to functions; **:** is the application operation; **D** is a set of function definitions.

CHIARA includes the special object $\perp$, atoms like integer fixed and floating-point numbers, characters and strings, and sequences denoted with angle brackets like

$$\langle 1, 2, 3 \rangle, \langle \langle 1, 2 \rangle, \langle 3, 4 \rangle \rangle$$

CHIARA peculiarity is that its elementary operators are the D# operators. In CHIARA they form the *functionally complete* set – able to generate any other more complex function a program may need by applying the metacomposition rule. The above definitions allow the translation of any CHIARA program in a D# program. A detailed description of CHIARA in terms of its objects, functions, functional forms, application operations, and function definitions can be found in [16]; so, in this paper we only refer to some example operators that are present in the *h*HLDS model.

The powerful program algebra of CHIARA is able to extract all parallelism a program has in terms elementary operators, and, after the translation in D#, it can be consumed by the configurable dataflow machine. In fact, since every function in CHIARA can be written as a composition of basic functions and functionals, e.g. something like:

$$def\ f = f_1 \circ f_2 \circ \ldots \circ f_k \circ \ldots \circ f_{n-1} \circ f_n$$

it often happens that some segments of this expression, $f_k \circ \ldots \circ f_{n-1} \circ f_n$ for some $k \leq n$, turn out to involve just routing functions, i.e. functions that move data between places without performing any kind of actual computation but only sequence-to-sequence or sequence-to-object transformations. In our prototype, combinatorial operators are turned into suitable DAC connections since they do not require "computational" resources – they only require suitably configured switches between the DACs computing the object they are applied to and the DACs actually consuming the result of this computation.

At the language level, this can be justified by the fact that CHIARA only admits single-valued functions, and therefore some segments of the function computation just move data to the functions that have to consume them. In addition, as CHIARA programs are variable free, we can easily recognize in the code the functions that only route data to the places where they are consumed and distinguish such code from the one that actually performs computations. In a framework where, to exploit dataflow parallelism for their execution, programs and data are distributed over several processing nodes, this point is very important because it facilitates the mapping and scheduling process. Indeed, we have the possibility to allocate first the initial data set over the dataflow execution engines and then to follow the well defined instructions present in the program.

The whole compilation process is composed of two steps (cf. Fig. 3(a)):

- a compilation step that translates in D# the program, creating thus the DPG;
- a mapping step that partitions the DPG and constructs the list of parts to assign to processing nodes.

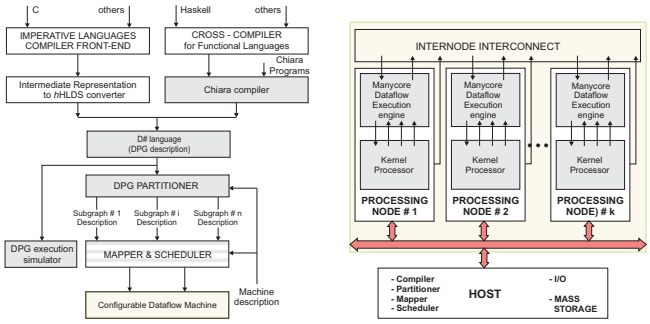
D# and DPG generation. Just to give an idea of CHIARA's expressive power, the typical way of writing CHIARA programs,

and how the overall compilation step works, let us consider as an example the program that evaluates the zeroes of a function using the bisection method. The CHIARA code is shown in Fig. 2(a). After the translation step, the compiler produces the D# code and its graphical representation as shown in Figs. 2(b) and 2(c). The next step involves generating the DPG table. The DPG table is in turn divided in two tables with two physically separated purposes: the former becomes the configuration graph table without any data (initial and constant) value, the latter becomes the initial and result- data with only the data values and the information needed to properly associate them to the actors which come from, and the locations where the Result Data. The importance of this phase is both to cut the D# code off from the mapper job and to allow the overlap of the configuration activities into the Kernel Processor (see Fig. 3(b) and Fig. 4(a)) and of the execution into the dataflow execution engine.

DPG mapping. We did not explore extensively here the problem of a general DPG mapping, which is left for future work. Here the aim is introducing the partiotioning problem and observing that in case of a large DPG out envisioned solution implies a proper scheduling in time on the available resources. The entire mapping process is built around the activities of the compiler (preparatory phase) and the activities of the mapper (executive phase), oriented to partition the configuration graph table and create list of sub-DPGs to assign to processing nodes. Before starting the partitioning, we need to point out some parameters that characterize the resources that the configurable dataflow machine offers. For the machine we have pinpointed these parameters that characterize resources: the number of DACs n_{DAC} in a MDE, the number of processing nodes N_{pn}, and the number of DACs N_{DAC} in all the configurable machine. Likewise, in terms of abstract entities, these parameters become: the number of actors N_a in a DPG, the number of actors $n_{N_{a_i}}$ in the *ith* subgraph, and the number of subgraphs N_{sg}. As far as the mapping and partitioning steps are concerned, either the DPG is larger than the available number of DACs or not. In the latter case, to solve the problems related to the third case shown in the table, it has been necessary to introduce the concept of temporal partitioning to partition a task into temporally interconnected subtasks. Computations represented by a DPG with requirements (problem size) that exceed the machine number of DACs and memory locations (machine configuration) cannot be completely mapped on the machine without using scheduling policies. Even if the topic of scheduling policies is not treated here, thanks to our architectural choices, the automatic scheduling of sub-DPGs is guaranteed for the execution of such a DPG, allowing thus the execution of DPGs with unlimited number of resource requirements (at least from the theoretical point of view).

IV. The configurable dataflow machine

A configurable dataflow machine has been developed within an integrated software-hardware project named Demand Data Driven Architecture System (D^{3}AS) [23]. Fig. 3.1 describes the toolchain. Gray boxes represent the software developed and the sequence of the entire compilation process that generates the DPGs expressed in D# language (III). White boxes represent



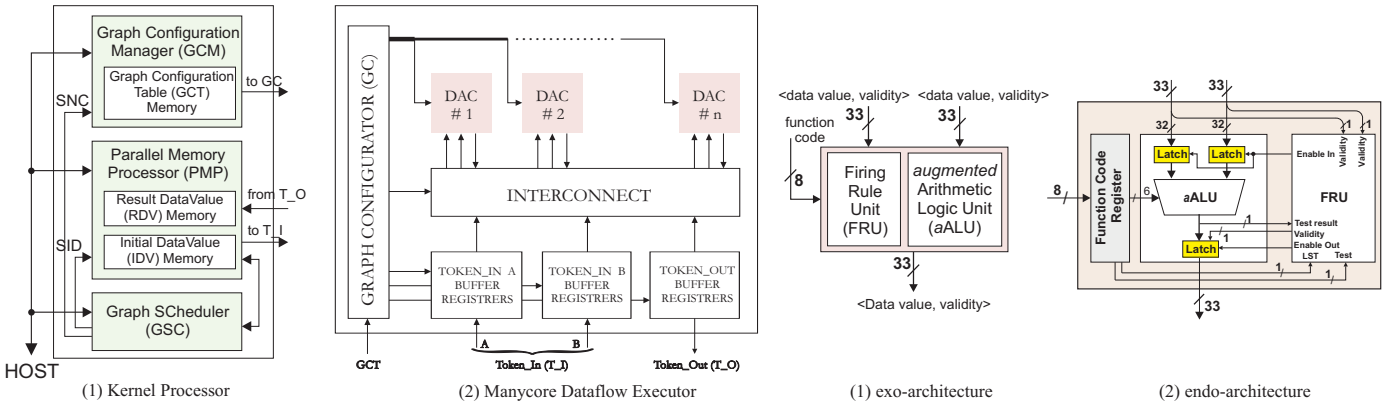
(a) general project (b) general architecture

Fig. 3: D^{3}AS computing system

the software to be developed at a later stage. Fig. 3(b) shows the general D^{3}AS architecture. It consists of a configurable dataflow system constituted by *k* processing nodes, an internode interconnection for the communication of DPGs allocated over more nodes and a host, to supply all software activities to compile, partition, map, and create the ordered list of DPGs to assign to each processing, mass storage, and etc..

The processing node architecture. Each processing node is constituted by a the Kernel Processor (Fig. 3(b) and Fig. 4(a).1) that supports the macro-functions to load a DPG onto the Many-DAC Dataflow Execution-engine (MDE) shown in Fig. 4(a).2. Instead of a von Neumann-based many-core chip, the MDE is a many-core chip where the *n* DACs constitute a set of completely decentralized and self-scheduling execution units connected through the interconnecs (a discussion on which type is more appropriate is in the next subsection *The Internode Interconnects*) that implements arcs and links. The graph configurator register holds the DAC function codes and the crossbar-switch code that constitute the DPG configuration loaded onto the MDE. Three I/O register banks to receive/send data tokens from/to the *Kernel Processor*. So, once configured, DACs can fire as soon as valid data tokens start entering the MDE thanks to their asynchronous behavior. A DAC (Fig. 4(b)) consists of a firing rule unit that implements the *h*HLDS firing rules by means a simple hardware circuitry, and an *augmented* ALU that implements the D# operator set. In this way, dataflow actor operations are sequenced by the data validity firing rule. In cooperation with the MDE, the Kernel Processor includes three fundamental blocks whose functionalities are:

- *Graph Configuration Manager* (GCM): (i) all the sub-DPG configuration tables are stored in the Graph Configuration Table memory; (ii) when an enabling Send-Next-Configuration signal from the Graph Scheduler reaches the graph manager, the graph manager transfers the scheduled graph configuration from its memory to the MDE.
- *Parallel Memory Processor* (PMP): at the same time, the scheduler sends the enabling Send-Initial-Data signal to this memory processor that (i) prepares the initial data tokens the for their transfer to the dataflow execution engine; after having organized previous transfer, (ii) prepares the result data token for tokens transfer from the dataflow execution engine to the output buffer, as soon as they are



(a) Processing Node internal architecures

(b) Dataflow Actor Unit details

Fig. 4: configurable Processing Node architecture

ready to the output buffer registers; when the computation ends, it (iii) sends a termination signal to the scheduler.

- *Graph SCheduler* (GSC): (i) it implements the scheduling policy (defined after the partitioning and mapping activities) for the sub-DPGs allocated on the processing node; (ii) it sends enabling signals to the graph configuration manager and to the parallel memory processor; it (iii) manages the interaction with the Host.

The Internode Interconnect. At the moment, we are evaluating three possible solutions. The reason why we have not chosen a solution yet, is that each of them conditions parameters like scheduling policy, scalability, cost, performance, and so on when the DPG dimension requires to be sub-DPGs over several processing nodes. For an example, from an architectural point of view a crossbar switch that connects the processing nodes would be the ideal solution to transfer data token between them because would offer the maximal flexibility and preserve the pure dataflow execution and the self-scheduling policy is the only one needs. However, this solution, in terms of scalability, complexity, and cost, results hardly feasible. On the other hand, a direct network topology would be a solution that offers a very high scalability and limited cost but no flexibility. However, this solution, in terms of performance like mapping, latency, bandwidth, routing algorithms is unacceptable also due to the fine grain operations that dataflow offers. Finally, a trade-off solution between the two extremes would be the best solution. However, our work on this solution is still not mature to present any reliable result.

From the architectural point of view, our configurable dataflow machine is an Interconnection Mode Configurable machine because in each MDE the storing of temporary data and instructions are eliminated as traffic. This because during the execution of a sub-DPG, neither instructions nor data need to be fetched from memory. In fact, once the sub-DPG configuration is loaded from the kernel processor onto the dataflow execution engine, its execution happens when the parallel memory processor loads the set of data tokens onto the MDE. So it is that the ”computing beat” depends on the data tokens availability, e.g. as it happen during a pipelined execution dataflow. That makes our architecture different from other dataflow and FPGA-based ones.

V. Evaluation metrics

Several issues affect the performance of current highest performance computers (listed in the Top500) when dealing with highly data-intensive workloads [24]. In contrast, current unconventional architectures (FPGA-based, dataflow, etc.) show a better speedup than Top500 machines. Also, same Authors [24] suggested to add new measurements to the current performance metric to take into account parameters, such as pin throughput, local memory size/bandwidth, power consumption – as asynchronous circuits consume less power. Moreover, such metrics are partially adopted by the Green500 list. They also underlined the issue in evaluating radically different models of computation, such as dataflow, that remains yet to be addressed. Following their advice, we have pinpointed some parameters that can be taken into account to evaluate a dataflow architecture like ours with a clockless many-DAC dataflow processor (MDE).

For the MDE we have first specified the evaluation model to identify which critical parameters condition the latency for loading a DPG configuration from the Kernel Processor to the dataflow execution engine, and then explain how to determine its computation time. Since the MDE versatility to execute different DPGs on it, this has been the critical part for the parameter identification. Regarding the power consumption evaluation, we decided here to focus on other metrics detailed below (not the power). Regarding the Kernel Processor we have not pinpointed any critical parameter because it is mainly a hardware controller that manages all necessary activities to execute the sub-DPGs assigned to the node processing and because it generally overlaps its activities with the MDE execution.

Evaluation model. Significant performance parameters that should be considered are: the latency t_{DAC} to execute a DAC operation; the latency t_{cnf} to load a sub-DPG configuration – network and DACs codes, and the latency t_{Itk} and t_{Otk} to load the initial and the result tokens, respectively, between the

TABLE I: MDE latency configuration (ns)

t_{conf}	t_{Ir-r}	$Tconf$
28	4	32

Kernel Processor and the dataflow execution engine. Regard to the execution time, we consider the latency for a acyclic sub-DPG, but only one execution of its if it is an iterative data-dependent graph. Obviously, the execution time $t_{e_{DPG}}$ of a sub-DPG depends on the number of sequential levels (steps) composing the loaded graph. As an example, let us consider the D# graphical representation shown in Fig. 2(c). Since the DPG is scattered in 28 operations organized in 12 sequential levels, the number of sequential levels n_{sql} establishes the execution time, $t_{e_{DPG}} = n_{sql} \times t_{DAC}$ if it entirely fits in an MDE.
Other parameters are the latency t_{Or-r} and t_{Ir-r} needed to transfer and stabilize bits between external registers and internal registers, respectively. t_{Or-r} occurs when a configuration and input tokens move from the Kernel Processor to the dataflow execution engine. Due to the limited pin number that a device has, the bit transfer to the dataflow execution engine is organized in nps pipeline stages whose buffer-size is the number of pins n_p used for the throughput thr to and from the dataflow execution engine. For the parameters, the independent variable are: n_p, the number of DACs n_{DAC} inside an MDE; the number of bits nb_{tk} to represent a data token; and the number of bits nb_{op} to represent an operation code.

To define the total number of bits required for the execution of a sub-DPG – the formulas for all the input tokens Nb_{Itk} and the configuration Nb_{cnf} (network and DACs codes) are:

$$Nb_{Itk} = (2 \cdot nb_{tk}) \cdot n_{DAC} \quad (1)$$

$$Nb_{cnf} = n_{DAC} \cdot (2\lceil log_2\ n_{DAC} \rceil + nb_{op}) \quad (2)$$

$$Nb_{tr} = n_{DAC} \cdot (2\lceil log_2 \cdot n_{DAC} \rceil + nb_{op} + 2 \cdot nb_{tk}) \quad (3)$$

$$nps = \lceil \tfrac{Nb_{tr}}{thr} \rceil \quad (4)$$

where Nb_{tr} is number of the bits required to load a sub-DPG in the dataflow execution engine. After that the evaluation model becomes:

$$t_{conf} = t_{Itk} = t_{Otk} = nps \cdot t_{O_{r-r}} \quad (5)$$

$$Tconf = t_{conf} + t_{Ir-r} \quad (6)$$

Latency evaluation. For the MDE estimate, we have used two Altera APEX 20K15-C, one for the Kernel Processor and one for the MDE, whose available pins we have exploited a number of pins $n_{pin} = 830$ for the communication with the Kernel Processor. For its realization, it has been possible to only carry out $n_{DAC} = 28$ interconnected DACs after several attempts due to the interconnect Area penalty that quickly consumed the device resources. Then we set $nb_{tk} = 33$ and $nb_{op} = 8$.
Once defined the MDE characteristics, the throughput between Kernel Processor and dataflow execution engine has been thus organized: 231 pins for each token set T_I(A), T_I(B), and T_O; 70 and 56 pins for the crossbar switch code and the DAC function codes, respectively, employing residual pins for control signals like reset internal I/O buffers, reset all, acknowledgment. The set of registers has been the trade-off result between a low nps value and a low I/O pin penalty – the more unused I/O pins, the higher pin penalty, then consequently the higher is nps.
With the Altera tools, we evaluated t_{Or-r}, t_{Ir-r}, and t_{DAC}. For them, we measured $t_{Or-r} = 7ns$, $t_{r-r} = 4ns$, and $t_{DAC} = 30ns$. As the interconnect latency turned out to be a small value (less than $1ns$), we ignored it. Applying the evaluation model formulas, Table I shows the latency that occurs when the MDE changes its configuration from a DPG to another. This are the preliminary results for our small example of Fig. 2. The whole process of configuring an FPGA for the same example may require an average of 7 minutes [25].

VI. Related works

The continuous growth of the number of transistors and cores per chip and a lack of significant advance in the conventional parallel processing arena have renewed an interest in dataflow computing. A major active dataflow project that is investigating on how to exploit program parallelism with many-core technology is TERAFLUX. Its challenging goal is to develop a coarse grain dataflow model to drive fine grain multithreaded or alternative/complementary computations employing Teradevice chips [7][8][10]. The project covers almost all aspects from the programming models in standard languages such as C, OpenMP, StarSs [26], OpenACC [27] through the full system simulation of standard cores running off-the-shelf Linux, where dataflow computation can be offloaded to dataflow accelerators. DPG could be potentially generated as output of the compilation toolchain similarly to Fig. 3(a). Since standard threads are seriously flawed because wildly nondeterministic and implementing a multithreaded computation model is difficult [28], an interesting idea proposed in TERAFLUX is *Dataflow Threads* (DF-Threads) [11][7]. A similar concept is the *dataflow codelet* or simply *codelet* [12]. A codelet is collection of machine instructions smaller than a thread but coarser than traditional dataflow and represents the finest granularity of parallelism that can be scheduled as a unit of computation. Its operational semantics asserts that a codelet is first enabled, when all its events are satisfied, and then fired (scheduled), when a processing element becomes available. DF-Threads and Codelets represents a step towards the fine granularity of dataflow. Both execution models (DF-Threads and Codelets) are still far from the effective and efficient self-scheduling of an actor present in the static model. More recently, Less recently, new reconfigurable architectures very similar to the dataflow approaches have been proposed. TRIPS [29] is based on a hybrid von Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor core with an adaptive on-chip memory system. TRIPS uses three different execution modes, focusing on instruction-, data- or thread-level parallelism. WaveScalar [30], on the other hand, totally abandons the program counter. Both TRIPS and WaveScalar take a hybrid static/dynamic approach to scheduling instruction execution by carefully placing instructions in an array of processing elements and then allowing execution to proceed dynamically. But, in our configurable dataflow machine during the execution of an algorithm it is not necessary to fetch any instruction or data from memory. To execute a DPG, all it needs from memory is the initial data-token set, so any data

structure can takes advantage of that to increase the transfer rate and overcome the problem of the inefficiency in handling big data, typical of the dataflow model. the execution of a DPG is based on the *homogeneous* High Level Dataflow System [14], a static dataflow model, whose peculiarity is the presence of only valid data-tokens and actors with homogeneous I/O conditions (only one output and two input arcs).
Another lively interest in dataflow comes from FPGAs-based computing. In this field, there is a lot of research on the mapping and execution of dataflow graphs [31] [32], but no solution is addressed to their execution in hardware without employing the control flow information. Besides, the reconfiguration time cost when a DPG changes is beyond compared to our MDE.

VII. Conclusions

This paper describes the Interconnection Mode Configurable Dataflow-Machine based on a novel approach to the architectural and programming models. It represents not only a drastic departure from conventional computer organizational concept but also a drastic departure from the other dataflow machines. A detailed description of the computing architecture is given together with the execution model and the graphic machine language D#, based on the *h*HLDS model. A re-examination of the functional programming style is also given. The whole compilation process, composed by a compilation step and a mapping step is described, and an evaluation model for the dataflow execution engine is also presented. Finally some preliminary results are reported on the configuration cost latency.

Acknowledgements

We wish to thank the anonymous reviewers for their useful comments. This work is partly supported by the European Project TERAFLUX (id. 249013).

References

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

[2] S. Dosanjh, R. Barrett, M. Heroux, and A. Rodrigues, "Achieving exascale computing through hardware/software co-design," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011, vol. 6960, pp. 5–7.

[3] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–25.

[4] M. Amamiya and R. Hasegawa, "Dataflow computing and eager and lazy evaluations," *New Generation Computing*, vol. 2, no. 2, pp. 105–129, 1984.

[5] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *ACM Comput. Surv.*, vol. 14, no. 1, pp. 93–143, Mar. 1982.

[6] M. Technologies, http:www.maxeler.com/technology/dataflow-computing/.

[7] R. Giorgi, "TERAFLUX: Exploiting dataflow parallelism in teradevices," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: ACM, 2012, pp. 303–304.

[8] R. Giorgi and et al., "Teraflux: Harnessing dataflow in next generation teradevices," *ELSEVIER Microprocessors and Microsystems*, Apr 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933114000490

[9] R. Giorgi, Z. Popovic, and N. Puzovic, "Implementing fine/medium grained tlp support in a many-core architecture," in *Proceedings of 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2009*. Samos, Greece: Springer, jul 2009, pp. 78–87.

[10] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer, "Architectural support for fault tolerance in a teradevice dataflow system," *Springer International Journal of Parallel Programming*, no. 0, pp. 1–25, May 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0312-y

[11] R. Giorgi and et. al, "Definition of isa extensions, custom devices and external cotson api extensions," Siena, Italy, pp. 1–78, mar 2011, deliverable. [Online]. Available: http://www.dii.unisi.it/ giorgi/papers/Giorgi11b.pdf

[12] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines: Position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69.

[13] G. Estrin, "Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer," *Annals of the History of Computing, IEEE*, vol. 24, no. 4, pp. 3–9, Oct 2002.

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

[15] ——, "Position paper: Validity of the static dataflow approach for exascale computing challenges," in *International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2013), in conjunction with PACT 2013*, Edinburgh, Scotland, UK, Sep. 8, 2013.

[16] L. Verdoscia, M. Danelutto, and R. Esposito, "CODACS prototype: CHIARA language and its compiler," in *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*. IEEE Computer Society Press, March 2004, pp. 864–870.

[17] J. Dennis, J. Fosseen, and J. Linderman, "Data flow schemas," 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. 187–216.

[18] R. Miller and J. Rutledge, "Generating a data flow model of a program," ser. IBM, T. D. Bulletin, Ed., 1966, vol. 8, pp. 1550–1553.

[19] 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.

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

[21] 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.

[22] S. Patil, "Closure properties of interconnections of determinate systems," in *The project MAC Conference on Concurrent Systems and Parallel Computation*. ACM, 1970, pp. 107–116.

[23] L. Verdoscia and R. Vaccaro, "D^3AS Project: A Different Approach to the Manycore Challenges," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: ACM, 2012, pp. 261–264.

[24] 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.

[25] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe, "Realistic performance-constrained pipelining in high-level synthesis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[26] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the openmp tasking model to allow dependent tasks," in *OpenMP in a New Era of Parallelism*. Springer, 2008, pp. 111–122.

[27] S. Bihan, "CAPS OpenACC Compilers: Performance and Portability. CAPS OpenACC Compilers: Performance and Portability," Feb 2013.

[28] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[29] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team, "Scaling to the end of silicon with edge architectures," *Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[30] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, pp. 4:1–4:54, May 2007.

[31] D. Capalija and T. Abdelrahman, "A coarse-grain fpga overlay for executing data flow graphs," in *The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2012)*, Portland, Oregon, June 10 2012.

[32] A. C. F. D. Silva, "The chipcflow project to accelerate algorithms using a dataflow graph in a reconfigurable system," *WSEAS Transactions on Computers*, vol. 11, pp. 265–274, 2012.