Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 3190234, 21 pages
http://dx.doi.org/10.1155/2016/3190234

Research Article

A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs

Lorenzo Verdoscia[1] and Roberto Giorgi[2]

[1]*Institute for High Performance Computing and Networking, CNR, 80131 Naples, Italy*
[2]*Department of Information Engineering and Mathematics, University of Siena, 53100 Siena, Italy*

Correspondence should be addressed to Lorenzo Verdoscia; lorenzo.verdoscia@na.icar.cnr.it

Received 23 December 2015; Revised 8 March 2016; Accepted 27 March 2016

Academic Editor: Zoran Obradovic

We present a new type of soft-core processor called the "Data-Flow Soft-Core" that can be implemented through FPGA technology with adequate interconnect resources. This processor provides data processing based on data-flow instructions rather than control flow instructions. As a result, during an execution on the accelerator of the Data-Flow Soft-Core, both partial data and instructions are eliminated as traffic for load and store activities. Data-flow instructions serve to describe a program and to dynamically change the context of a data-flow program graph inside the accelerator, on-the-fly. Our proposed design aims at combining the performance of a fine-grained data-flow architecture with the flexibility of reconfiguration, without requiring a partial reconfiguration or new bit-stream for reprogramming it. The potential of the data-flow implementation of a function or functional program can be exploited simply by relying on its description through the data-flow instructions that reprogram the Data-Flow Soft-Core. Moreover, the data streaming process will mirror those present in other FPGA applications. Finally, we show the advantages of this approach by presenting two test cases and providing the quantitative and numerical results of our evaluations.

1. Introduction

There is still a slight inclination of part of the High-Performance Computing (HPC) community to embrace the data-flow ideas in order to speed up the execution of scientific applications. The reasons are mostly of a pragmatic nature rather than technical [1–4]. A dominant reason why the HPC community and, in particular, the applications programmers do not pay more attention to the advanced data-flow architecture ideas is due to the fact that, in the past, very high-performance data-flow systems of commercial grade were not readily available on the market. Simulations and relatively low speed, low density academic prototype, and performance inefficiencies did not make data-flow architectures attractive to computational scientists [5, 6] because they did not offer the opportunity to application programmers to run their problems faster than before. However, despite general scepticism for past disappointing results, it is coming out that data-flow systems are still a valid manner to increase performance [1]. These systems, employing Field Programmable Gate Array (FPGA) (readily available on the market, nowadays [7]) to implement data-flow accelerators, outperform most of the TOP 500 supercomputers not being paradoxically included in the list [8–10]. This happens because the (re)configurable computing paradigm offers a performance of custom hardware and flexibility of a conventional processor [11–13]. Because of this flexibility and the intellectual property availability, the (re)configurable approach does not only significantly accelerate a variety of applications [14] but also constitutes a valid execution platform to form programmable high-performance general purpose systems [15]. In particular, given its fine grain nature, the static data-flow execution model is promising when applied to this platform [3, 16, 17].

Spatial reconfigurable computing, such as FPGAs, massively parallel systems based on soft-cores, coarse-grained reconfigurable arrays (CGRAs), and data-flow-based cores, accelerates applications by distributing operations across many parallel compute resources [18]. Nowadays, FPGAs constitute a formidable tool for prototyping more complex reconfigurable and general purpose soft-cores, where the

most recent ones not only incorporate Digital Signal Processor (DSP) capabilities but also address the interconnect issue, the number one bottleneck to system performance at advanced nodes, although FPGAs continue to retain their primary characteristic of being bit-programmable.

The use of soft-core processors in building parallel systems brings in many advantages such as flexibility, the possibility to be synthesized almost for any given target Application-Specific Integrated Circuit (ASIC) or FPGA technology, the possibility to describe functions through higher abstraction levels, by using an Hardware Description Language (HDL), and many more. However, compared to custom implementations, soft-cores have the disadvantages of larger size, lower performance, and higher power consumption [19].

CGRAs ([15, 18, 20] give noteworthy surveys) consist of reconfigurable processing elements (PEs) that implement word-level operations and special-purpose interconnects retaining enough flexibility for mapping different applications onto the system. The reconfiguration of PEs and interconnects is performed at word-level too. CGRAs offer higher performance, reduced reconfiguration overhead, better area utilization, and lower power consumption [21] compared to fine-grain approaches. However, CGRA architectures present several limits. Firstly, because they mainly execute loops, CGRAs need to be coupled to other cores on which all other parts of the program are executed. In some designs, this coupling introduces run-time and design-time overhead. Secondly, the interconnect structure of a CGRA is vastly more complex than that of a Very Long Instruction Word (VLIW). Finally, programmers need to have a deep understanding of the targeted CGRA architectures and their compilers in order to manually tune their source code. This can significantly limit programmer productivity [22].

In the past, Miller and Cocke [11] proposed a new class of configurable computers, interconnection mode and search mode. In contrast with a von Neumann-based machine, these machines configured their units to execute the natural and inherent parallelism of a program after exposing it like a data-flow graph. Because of their configurable unit organizations, the configurable search and interconnection modes have constituted the basic models of data-flow machines [23]. Although there are several data-flow architectures proposed, most of them fall into the search mode configurable [24]. Overall, only one can be classified as partially of the interconnection mode type and as partially of the search mode type [25]. Differently, the Data-Flow Soft-Core processor falls into the interconnection mode configurable machines. Our approach differs from strengthened reconfigurable computing. For example, in a CGRA, once mapped, a data-flow graph is executed like what happens in a data-flow schema [26] by means of the associated control flow; in an FPGA, the loading of a new data-flow graph needs a configuration bit-stream that requires, at best (partial reconfiguration), at least a delay of tenths of μs [27]. Conversely, in our case, not only data actually flow among actors without any associated control flow through a custom crossbar-like interconnect, but also the full reconfiguration time, for a new data-flow graph, requires only a delay of a few dozen ns [17].

While configurable computing has revealed its effectiveness over parallel systems based on conventional core processors [1], how to efficiently organize resources available at 14 nm technology or less, in terms of programmability and low power consumption, remains an open question [28]. Here we discuss a new concept of soft-core that can be effectively and efficiently supported by FPGAs with adequate interconnect resources called the "Data-Flow Soft-Core" (hereinafter DFSC) processor.

The idea is to make available, on a reconfigurable chip, a processor that accelerates data processing after loading data-flow instructions rather than control flow instructions. Data-flow instructions, which come out from the demand-data-driven codesign approach [29], serve here both to describe a program and to change the structure of the data-flow accelerator, without need of a partial or full reconfiguration. Our design aims at providing the performance of an interconnection mode data-flow architecture and the flexibility of reconfiguration, without having to pass a new bit-stream for reprogramming the DFSC processor. We are going to show the advantages of this approach by presenting some examples and a test case providing their numeric evaluations.

Our main contributions can be, thus, summarized as follows:

(i) data-flow implementation of a certain function or functional program which can be exploited by simply relying on its description through the data-flow instructions that will reprogram the DFSC;

(ii) inside the DFSC, data streaming occurring in a similar way as in other reconfigurable computing applications;

(iii) the DFSC data-flow accelerator (shortly referred to as accelerator) which can be reprogrammed to implement a new data-flow program graph (DPG), which represents the new program, by switching its contexts, sub-DPGs fitting (containable) into the accelerator, on-the-fly without the need of any bit-stream reconfiguration;

(iv) minimal set of instructions to execute a data-flow program;

(v) elimination of both temporary data and instructions as traffic over the memory access busses.

The remainder of this paper is organized as follows. Section 2 presents the DFSC ISA; Section 3 discusses related work in this area; Section 4 describes the DFSC architecture and explains the software toolchain; in Section 6 we discuss a test case based on matrix multiplication with some results; Section 7 highlights the main differences between some CGRAs and the DFSC architecture; Section 8 provides our conclusions.

2. The DFSC Processor Instruction Set

In contrast to a conventional soft-core processor that is mainly based on a RISC architecture, the DFSC processor has a custom architecture derived from the codesign process

between the functional programming and the data-flow execution principles, given their strict relationship. In fact, the former can create the DPG by demanding a function for its operands (lazy evaluation) driven by the need for the function values. The latter can execute the DPG in data-flow mode by consuming operands (eager evaluation). In our case we used the functional language Chiara [30], based on the Backus FP programming style [31], together with the *homogeneous* High Level Data-Flow System (*h*HLDS) model [32].

2.1. hHLDS Overview. The High Level Data-Flow System (HLDS) [32] is a formal model, which describes the behavior of directed data-flow graphs. In this model, nodes are actors (operators) or link-spots (places to hold tokens) that can have heterogeneous I/O conditions. Nodes are connected by arcs from which tokens (data and control) may travel, whereas *h*HLDS describes the behavior of a static data-flow graph imposing homogeneous I/O conditions on actors but not on link-spots. Actors can only have exactly one output and two input arcs and consume and produce only data tokens; link-spots represent only connections between arcs. Since *h*HLDS actors do not produce control tokens, merge, switch, and logic-gate actors defined in the classical data-flow model [26] are not present. While actors are determinate, link-spots in *h*HLDS may be not determinate. In *h*HDLS there exist two types of link-spots: (i) joint, which represents a node with two or more input arcs and one output arc (it makes the first incoming valid token available to its output), and (ii) replica, which has only one input arc and two or more output arcs (it replicates its incoming token on each output arc). Joint and replica can be combined to form a link-spot with more input and output arcs. Despite the *h*HLDS model simplicity, it is always possible to obtain determinate DPGs including data-dependent cycles (proofs are given in [32]). Moreover, the model also simplifies the design of the accelerator with respect to the classical model as shown in Appendix A. The main features of *h*HLDS can be, then, summarized as follows:

(i) Actors fire when their two input tokens are valid (validity, an intrinsic characteristic of a token in the *h*HLDS model, is a Boolean value whose semantics is as follows: 1 (valid): the token is able to fire an actor; 0 (not valid): the token is unable to fire an actor), and no matter if their previous output token has not been consumed. In this case, the actor will replace the old output token with the new one. In a system that allows the flow of only data tokens, this property is essential to construct determinate cycles (data-dependent loops).

(ii) To execute a program correctly, only one way token flow is present as no feedback interpretation is required.

(iii) No synchronization mechanism needs to control the token flow; thus the model is completely asynchronous.

In *h*HLDS, actors and link-spots are connected to form a more complex DFGs, but the resulting DFG may be not determinate if cycles occur because no closure property can be guaranteed [33]. This happens for sure when the graph includes joint nodes, which are not determinate. When the DPG results to be determinate, we name it macroactor (mA). Obviously, an mA is characterized by having I (mA) > 2 and O (mA) ≥ 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).

2.2. The D# Assembly Language. The DFSC processor offers programming in a custom assembly language that is also the graphical representation language that describes the data-flow graph of a program. It has been defined applying the demand-data-driven approach to codesign methodology [29] between the functional paradigm and the *h*HLDS paradigm. Since macroactor 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, and IT_R mAs).

The Macroactor TEST. The simplest relational 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 Figure 1(a). If $a, b, c \in \mathbb{R}$, its semantics is

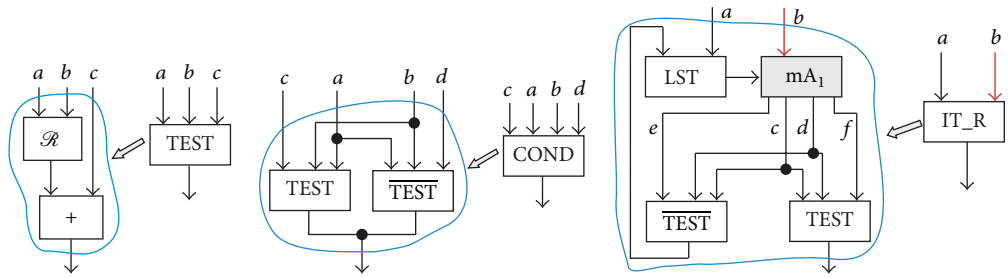
$$\text{TEST}\,(a, b, \text{c}) = \begin{cases} c & \text{if } a\,\mathcal{R}\,b \text{ is satisfied} \\ \perp & \text{otherwise.} \end{cases} \quad (1)$$

$\perp$ (bottom) stands 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 (our choice is 0) and the validity "not valid," in other words, conceptually absent.

The Macroactor COND. The simplest conditional structure is the mA COND, shown in Figure 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-spot Joint. If $a, b, c, d \in \mathbb{R}$ and $p = a\,\mathcal{R}\,b$, its semantics is

$$\text{COND}\,(a, b, c, d) = \begin{cases} c & \text{if } a\,\mathcal{R}\,b \text{ is satisfied} \\ d & \text{otherwise.} \end{cases} \quad (2)$$

The Macroactor 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 $\overline{\text{TEST}}$, a macroactor mA_1 or an arithmetic actor, and the actor LST (loop start) as shown in Figure 1(c). The LST



(a) TEST (b) COND (c) IT_R

Figure 1: The basic macroactors in D#.

semantics is the following: at the first firing the right token is selected, while the left token is selected for the subsequent firings. If $a, b, c, d, e, f \in \mathbb{R}$, its semantics is

$$\mathtt{IT\_R}\,(a, b, \mathrm{mA}) = \begin{cases} \mathtt{IT\_R} & \text{if } c\,\mathscr{R}\,d \text{ is satisfied} \\ f & \text{otherwise.} \end{cases} \tag{3}$$

Observing Figure 1(c), we point out that if mA_1 is an IT_R macroactor as well, the figure represents a determinate and well-behaved nested-data-dependent iterative structure.

D# Definition. D# programming system is a tuple (A, T, 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 D# language a program is a collection of standard instructions named *expressions* that form a DPG. Each expression refers to an actor and specifies its functionality. It is so organized:

$$\langle \mathbf{a} \rangle, \langle \mathbf{f} \rangle, \langle \mathbf{t}_L \rangle, \langle \mathbf{t}_R \rangle, \langle \mathbf{d}_O \rangle, \tag{4}$$

where **a** is the identifier number of the actor, **f** is the operation that the actor has to perform, $\mathbf{t}_L$ and $\mathbf{t}_R$ are the left and right input tokens, and $\mathbf{d}_O$ is the identifier of the actor number/numbers that has/have to receive the operation result. If the result is a final one, $\mathbf{d}_O$ is tagged *out*. If $\mathbf{d}_O$ is a list of integers separated by the - (dash) character, each corresponding actor in the list will receive the value produced. Regarding $\mathbf{t}_L$ and $\mathbf{t}_R$, 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 a value is consumed, it becomes not valid; if it is known only at run-time, for example, produced by an external event, then it is represented with a marker of two % characters; if it is a constant value of the program, its value ends with the % and remains valid until the context does not change.

Internal Data. It is the value that an actor produces for another actor. It remains valid until the producer does not fires again; it is an integer that represents the identifier number of the producing actor.

Table 1: DFSC operator set.

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

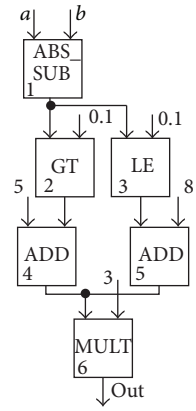
All the identifier actor numbers present in D# notation play the basic role to correctly and simply generate the code for the configurable network inside the data-flow execution engine and to allow using available software tools that can efficiently carry out the mapping phase.

2.3. The Elemental Operator Set. As a result of the codesign process between D# and *h*HLDS, we have determined a set of elemental operators which is functionally complete in the sense of the Backus FP style; more complex functions (higher order) are created by applying the metacomposition rule (combining primitive operators it is possible to change small programs into larger ones and produce new functions by applying functionals) and are consistent with *h*HLDS. Consequently, a program written in functional language consistent with the FP style, which includes this set, can be always translated at first hand in D#. Then the corresponding DPG can be directly executed by the DFSC processor. Table 1 shows the set of elemental operators. As it can be observed, this set does not include logical operators because *h*HLDS does not admit control tokens. However, their functionality can be expressed by higher order functions. The special operator ABS_, prefixed with an arithmetic operator, produces the absolute value of the corresponding operation. LST is the loop start operator employed in data-dependent cycles, and SL and SR are the operators that select the left and right actor tokens, respectively.

As an example of a code in D# language, 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`, token `5` is selected, otherwise token `8`. Finally, the result is scaled by a factor of `3`. The code and its graphical representation are shown in Figures 2(a) and 2(b), respectively. We would like to point out that the two actors GT (greater) and LE (less or equal) are mutually exclusive, so only

Node #	Func.	Left input	Right input	Output
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)

Figure 2: Example of a program in (a) D# language and (b) DPG graphical representation.

an actor that satisfies the predicate produces a valid token whose value is 0, while the other produces a nonvalid (don't care) token. This feature simplifies the design of the DFSC accelerator making it possible to use only identical data-flow functional units (DFUs) and only data wires to connect them. During the translation of a D# 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 the operation that an actor has to execute and its role in the DPG; and the *input token-value code*, which defines the input values that the actor has to receive in order to initiate 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.

When a DPG (the abstract entity in *h*HLDS) is loaded into the DFSC, it happens that (1) each DPG actor (abstract entity) is turned into one DFU (physical entity) such that the actor firing rules become the data-flow functional unit activation rules; (2) each arc/link-spot (abstract entity) 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 data-flow functional units; (3) each token, that is, its data and validity (abstract entity), is turned into a data value and its validity signal (physical entity) so that the self-scheduling of a DFU can happen.

3. Related Work

There exist several researches that investigate new architectural proposals for data-flow processors using FPGA as computation model. However, our data-flow machine is unique with respect to them because its reconfigurable processor executes data-flow program graph contexts only modifying the code of a custom interconnection and the operation codes of the computing units, that is, the actors of the data-flow program graph.

A major recent data-flow project that investigated how to exploit program parallelism with many-core technology is TERAFLUX [34–39]. Its challenging goal was to develop a coarse grain data-flow model to drive fine grain multithreaded or alternative/complementary computations employing Teradevice chips [40, 41]. However, the project did not address aspects on how to directly map and execute data-flow program graphs and how to tackle the dark silicon risk, but it rather introduced the concept of data-flow threads, DF-threads, and their memory model [42], which permits the execution of data-flow programs that also use shared-memory.

Among less recent, but still interesting FPGA-based reconfigurable architectures, we only considered those similar to our data-flow machine. TRIPS architecture [43] is based on a hybrid von Neumann/data-flow 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. The WaveScalar architecture [44], 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. However, in our configurable data-flow machine during the execution of an algorithm, it is not necessary to fetch any instruction or data from memory. The GRD (Genetic Reconfiguration of DSPs) chip [45] is specialized for neural network applications. It is constituted by a RISC processor to execute sequential tasks and 15 DSP processors to execute special tasks, connected in a reconfigurable network of a binary-tree shape. In contrast, the data-flow processor can execute both sequential and special tasks and its interconnect is organized like a crossbar. The MorphoSys chip [46] is constituted by the 8×8 RC Array, an array of reconfigurable cells (SIMD coprocessor) and its context memory, a TinyRISC main processor that executes sequential tasks, and a high-bandwidth memory

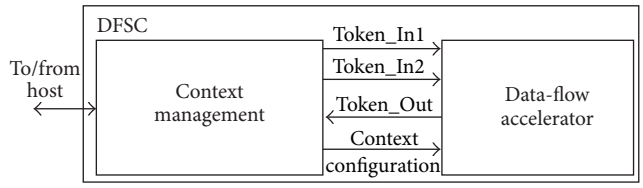


Figure 3: The Data-Flow Soft-Core (DFSC).

interface. Furthermore, it uses a 2D mesh and a hierarchical bus network. In contrast, our processor exhibits MIMD functionality, its interconnect is like a crossbar, and its context switch is managed by the context management according to the operations to be executed. For pipeline operations, the context does not change. The FPPA (Field Programmable Processor Array) processor [47] implements a synchronous fixed-point data-flow computational model. It employs 16 reconfigurable processing elements (PEs), a programmable interconnect, four 16-bit-wide bidirectional input/output ports, and one 16-bit-wide dedicated output port. The FPPA works in two phases: configuration, where PEs and programmable interconnects are configured to a specific behavior and to form a processing pipeline, and execution, where the program memory specifies sequences of PE and IO module "firings" individually. In the execution phase, the FPPA reads and processes the input stream of data and writes the result to the programmed output ports. The asynchronous data-flow FPGA architecture [48] describes a low-level application logic using asynchronous data-flow functions that obey a token-based compute model. In this FPGA architecture operators present heterogeneous I/O actors and they operate at cell rather than at computing unit level. Consequently, if the data-flow graph changes, they need a new reconfiguration string. Differently, since all computing units show homogeneous I/O conditions, in our processor, the context switching of a new data-flow program graph only requires the change of the operation and interconnect codes. The WASMII [49] system employs a reconfigurable device to implement a virtual hardware that executes a target data-flow graph. A program is first written in a data-flow language and then translated into a data-flow graph. The partitioning algorithm divides the graph into multiple subgraphs so that deadlock conditions cannot occur. However, the direct mapping of nodes and link-spots which executes a data-flow graph requires the reconfiguration of the device. In contrast, our data-flow machine differs from WASMII because the one-to-one correspondence between actors and computing units and arcs and physical connections happens simply by sending the operation codes to the computing units and the configuration code to the custom interconnect.

4. The Data-Flow Soft-Core Architecture

Several reasons shaped the design of the Data-Flow Soft-Core (DFSC). First, we wanted to map data-flow graphs onto hardware in a more flexible way than traditional HLS tools [50] 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 operations 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 DFSC consists of two main modules (Figure 3) as detailed in the following:

(i) data-flow accelerator (shortly accelerator), dedicated to executing DPG contexts;

(ii) context management, dedicated to managing the DPG contexts and/or the data tokens list for execution on the accelerator.

4.1. The Accelerator Architecture. The accelerator is composed of a DF-Code memory, a custom crossbar switch (DFU interconnect), and n_d identical DFUs. Figure 4 refers to an accelerator with $n_d = 64$ (64 represents a possible instance: the actual number of DFUs, the associated DFU interconnect, and the DF-Code memory can vary according to the available on-chip resources).

DF-Code Memory. This memory stores the DPG configuration (context) ready for execution. The *DFU interconnect code* memory is a register bank that is dedicated to storing the interconnect code ($2 \times n_d \times \lceil \log_2 n_d \rceil$) bits. The *DFU operating code* is a register bank that stores the DFU operating codes ($n_d \times 10$-bits). 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.

DFU Interconnect. The DFU interconnect, shown in Figure 6, consists of a custom crossbar grid of wires connected by switching elements that allow for the connection of any DFU output to any DFU input, except itself, or to the parallel memory processor (PMP) in the management module (see next subsection). All switches along a column are controlled by a $\lceil \log_2 n_d \rceil$-to-n_d decoder. Across a row only a valid token can exist because if two or more switches are enabled, they belong to some relational operation. But, in the *h*HLDS the operation is constituted by two or more actors in mutual exclusion; only the actor which satisfies the condition can generate the valid token. This feature is essential for implementing conditional and cyclic structures in conformance with the *h*HLDS model. When a decoder receives its own code, it enables the connection between the corresponding units or the PMP. The decoder control-signals come directly from the dedicated registers of the DFU interconnect code memory.

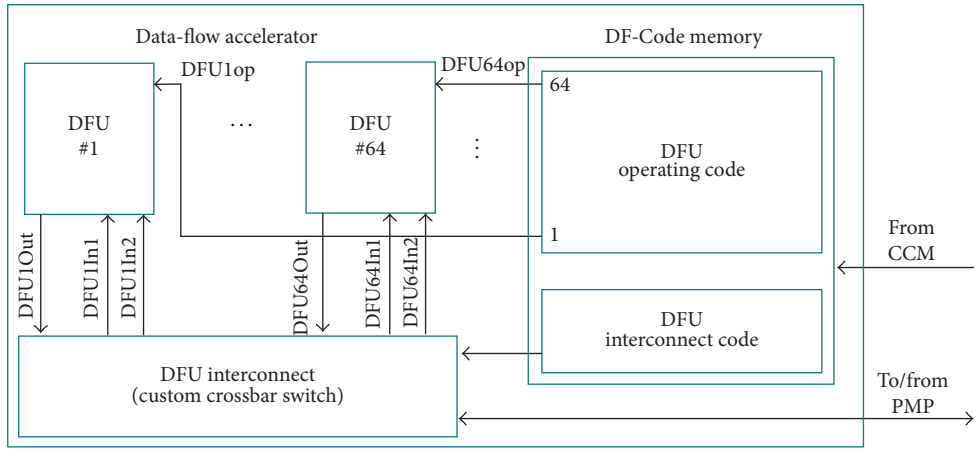


Figure 4: The accelerator module.

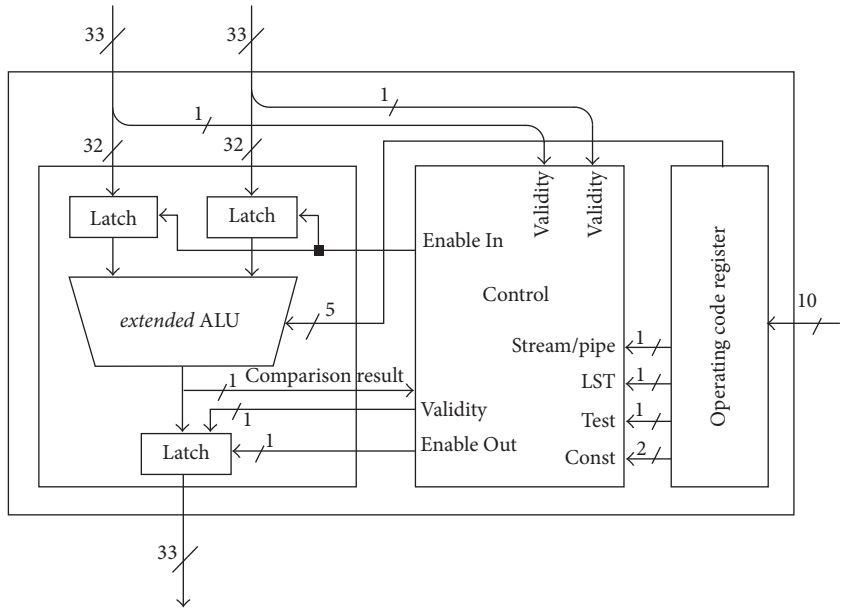


Figure 5: The DFU datapath.

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, in a recent work [51], the authors have shown that it is possible to implement a crossbar interconnecting 128 tiles with an area cost of 6% of the total.

Data-Flow Functional Unit. A DFU (architecture shown in Figure 5) implements any *h*HLDS actor, as defined in Section 2.1. 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 a 32-bit fixed-point *extended* ALU (arithmetic and comparison, multiplier, and divider) that implements the operator set and a 10-bit *operating code* register, which holds 5 bits for the operations and 5 bits for the DFU context (constant token, token streaming, loop, pipelining, and conditional participation). The control unit ensures the right behavior of an *extended* ALU (*e*ALU).

Control Unit and DFU Operation. When a valid input data token reaches the DFU, the control unit catches its validity bit to match the partner operand. As soon as the match occurs, an enabling signal activates the input latches that acquire the values of two input data tokens (DFUIn1 and DFUIn2) so that the operation stored in the register can take place; we call this *self-scheduling* of the operator. After the (fixed) known time for the *e*ALU operation, the control unit generates the validity bit for the output token, enables the output latch making available the result token, and resets the values of the two validity bits previously caught. The latches also isolate the internal DFU activities from the activity of other DFUs.

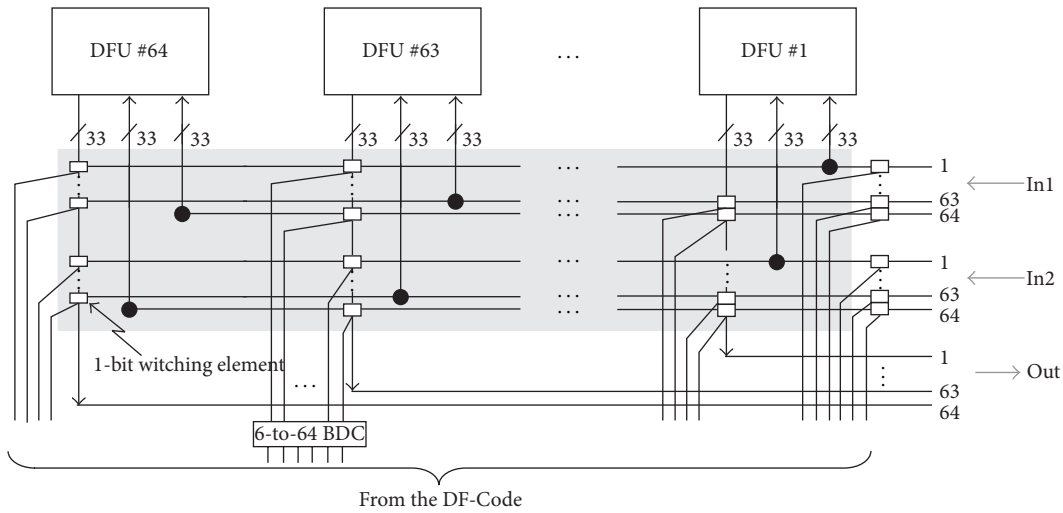


Figure 6: DFU interconnection network.

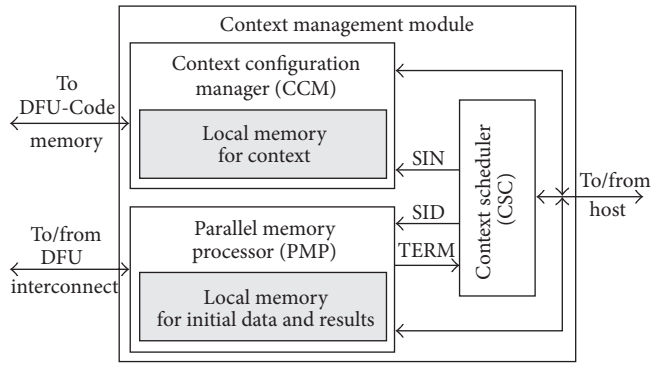


Figure 7: The management module.

Afterwards, a new firing process can start. The control unit also receives the five context bits for the *e*ALU from the operating code register.

Operating Code Register. This register holds 5 bits. The first bit, when set to 1, informs the control unit that the *e*ALU will work in pipelining fashion because of token streaming. The second bit, when set, informs the control unit that the *e*ALU is executing an LST operation. This is necessary because LST is the only operator fired by one token. In this case, since an *e*ALU only executes binary operations, as soon as a single valid token is present at one DFU input, the control unit generates a dummy presence bit for the other input so that the LST operation can start. The third bit (test bit), when set, informs the control unit that the *e*ALU is executing a comparison operation. If the condition is not satisfied, the control unit receives this information and resets token validity bit so that any other related DFU that follows it cannot fire. The last two bits—one for each input token—inform the control unit, when set, that the related data token will be reused. In this case the control unit sets the corresponding validity bit immediately after the reset signal.

4.2. The Context Management Module Architecture. It is constituted by three fundamental submodules (Figure 7).

(i) Context Configuration Manager (CCM). Once the contexts (i.e., the graph configuration) 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 (see below).

(ii) Parallel Memory Processor (PMP). While the CCM takes care of the program graph, this module takes care of the initial input data and collects the final output data that are processed by the accelerator. Therefore, once the scheduler enables the SID signal (Send-Initial-Data) in this submodule, the following actions are performed: (i) the initial data tokens are prepared to be transferred to the accelerator module; after having organized this transfer, (ii) the result data tokens are collected as soon as they are ready at the output buffer registers; when the computation ends, (iii) it sends a termination signal to the scheduler.

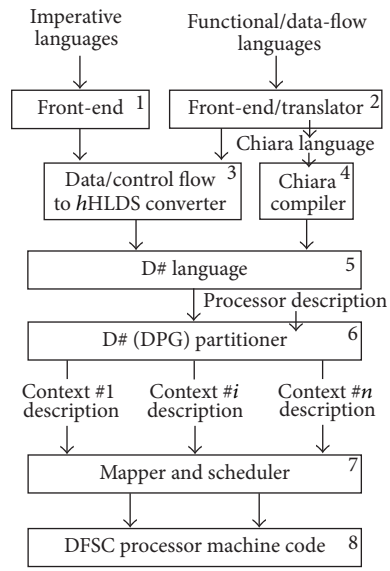


Figure 8: The programming toolchain for the Data-Flow Soft-Core processor.

```
// Matrix multiplication C(2, 2) = A(2, 2) × B(2, 2)
// with a11 = 1, a12 = 2, a21 = 3, a22 = 4
// with b11 = 5, b12 = 6, b21 = 7, b22 = 8

// DP Dot Product
 def DP = ! + ∘ & ∗ ∘ trans

// the matrix multiplication function
&& DP ∘ & distl ∘ distr ∘ [1, trans ∘ 2]: ⟨⟨⟨1, 2⟩, ⟨3, 4⟩⟩, ⟨⟨5, 6⟩, ⟨7, 8⟩⟩⟩
 stop
```

Box 1: Chiara code for the matrix multiplication $C(2, 2) = A(2, 2) \times B(2, 2)$.

(iii) Context Scheduler (CSC). Context scheduler (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 (see above) and to the PMP (see above) parallel memory processor, and (iii) manages the interaction with the host.

5. The Programming Toolchain

To turn programs into DPGs suitable for execution on the DFSC processor, we have developed some software tools, represented by blocks 4–8 of the toolchain of Figure 8. Blocks 1–3, under development, are dedicated to turning applications written in high level languages into graph contexts that the DFSC can execute. Here we only summarize the functionality of blocks 6–8, which is mostly beyond the scope of this paper.

(1) Block 6 partitions a DPG in contexts according to the number of DFUs inside the accelerator.

(2) Block 7 maps contexts onto the DFSC (or more DFSCs, if available) and creates the scheduling list for the context scheduler of the management module.

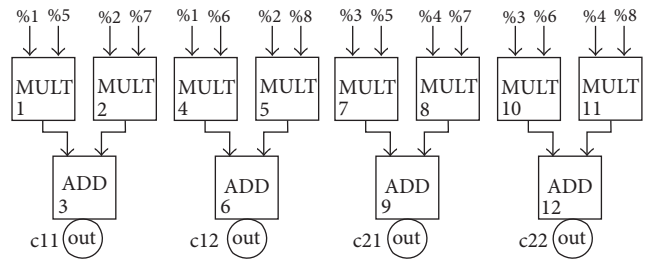
(3) Block 8 generates the DFSC machine code.

To test the toolchain, we used Chiara language because it maps more directly onto the DFSC assembly language and has combinator operators which are compiled into suitable DFU connections at the level of DPG. An example of Chiara program code for the matrix multiplication is shown in Box 1, while Figure 9 shows D# code.

The program matrix multiplication has four steps, tied by ∘ like $f \circ g$ functions, reading from right to left. Each step is applied in turn, beginning with $[1, \text{trans} \circ 2]$, to the result of its predecessor. The function dot product DP has three steps operating on conceptual units, as well, and no step is repeated. Moreover, this matrix multiplication program describes the essential operations of matrix multiplication without accounting for the process or obscuring parts of it

Node #	Func.	Left input	Right input	Output
1	MULT	%1	%5	3
2	MULT	%2	%7	3
3	ADD	1	2	Out
4	MULT	%1	%6	6
5	MULT	%2	%8	6
6	ADD	4	5	Out
7	MULT	%3	%5	9
8	MULT	%4	%7	9
9	ADD	7	8	Out
10	MULT	%3	%6	12
11	MULT	%4	%8	12
12	ADD	10	11	Out

(a)



(b)

Figure 9: Matrix multiplication example: (a) DFSC assembly language and (b) graphical representation.

```
#include <math.h>
float fun (float x){return (x∗x+3∗x - 1.75);}
int main(){
   float xmd, a= -1.34, b=1.0, epserr=1e-6, fm;
   do{ xmd=(a+b) / 2; fm=fun(xmd); if (fabs(fm) < epserr){ return 0;}
                          else{if (fun(a)∗fm < 0) b=xmd; else a=xmd;}}
while (fabs(fm) >= epserr); return 1;}
```

Box 2: Function root-finding with the bisection method: C code.

and yields the product of any pair $\langle m, n \rangle$ of conformable matrices.

Readers unfamiliar with functional programming languages can make reference to Appendix B for more details on the Chiara language and the matrix multiplication code.

6. DFSC Evaluation

In a recent paper [1], Flynn et al. argued that unconventional models of computation, for example, computation systems developed by Maxeler Technologies [7], when dealing with highly data-intensive workloads, show a better speedup than computers listed in the Top 500, but these systems do not appear in the Top 500 list. Also, their perspective about the performance is that metric should become multidimensional, measuring more than just FLOPS, for example, performance per watt, performance per cubic foot, or performance per monetary unit. However, the issue to evaluate radically different models of computation, such as data-flow, remains yet to be addressed. The reason is because, for custom data-flow systems, the current performance metrics do not take into account parameters, such as less power consumption, pin throughput, and local memory size/bandwidth.

Our DFSC processor falls into the asynchronous data-flow category. We recall that the DFSC processor is totally asynchronous, does not store partial results during the accelerator run, and employs an ad hoc memory in the context management, which acts differently from a cache, since it manages the execution of contexts in the accelerator. Here we evaluate the proposed architecture for two cases: cyclic and acyclic as reference examples. The system can change the program without uploading a new bit-stream (differently from classical FPGA accelerators). To show the potential of the DFSC, we used two simple but quite different algorithms—the bisection method, for finding a function roots, and the matrix multiplication—cyclic (data-dependent), the former, and acyclic, the latter.

6.1. Bisection Method Root-Finding. The bisection method for finding roots of a function represents an example of data-dependent iteration, where the cyclic flow of data is the only algorithm requirement. Given a function $f(x)$, continuous on a closed interval $[a, b]$, such that $f(a) \times f(b) < 0$, then the function $f(x)$ has at least a root (or zero) in the interval $[a, b]$. The method calls for a repeated halving of subintervals of $[a, b]$ containing the root. The root always converges, though more slowly than others.

For this algorithm, we only made a qualitative evaluation between the assembly codes for an x86 processor (the C program in Box 2) and D# code for the DFSC. The main reason is that a comparison time could not be fair due to the different execution times of a cycle for both x86 technologies (from Intel Pentium Dual CPU at 2 GHz to Core(TM)-i7 at 2.76 GHz) and the DFSC technology (180 μm and 32 ns for the execution time of a DFU). In fact, the run of the algorithm code in Box 2 for finding the root requires 22 cycles, while the execution times for the Dual CPU, the Core(TM)-i7, and the DFSC are 29 ms, 27 ns, and 8.5 μs, respectively. However, to give a sense of what happens with our processor, we report some qualitative evaluation. Observing the two low-level codes shown in Box 3 and Figure 10, a first interesting point is the simplicity and intelligibility of each D# code

```
    .file   "bisection2.c"
    .section
    .text.unlikely,"ax",@progbits
.LCOLDB3:
    .text
.LHOTB3:
    .p2align 4,, 15
    .globl  fun
    .type   fun, @function
fun:
.LFB33:
    .cfi_startproc
    flds    4(%esp)
    fld     %st(0)
    fmul    %st(1), %st
    fxch    %st(1)
    fmuls   .LC0
    faddp   %st, %st(1)
    fsubs   .LC1
    ret
    .cfi_endproc
.LFE33:
    .size   fun, .-fun
    .section    .text.unlikely
.LCOLDE3:
    .text
.LHOTE3:
    .section    .text.unlikely
.LCOLDB13:
    .section
    .text.startup, "ax",@progbits
.LHOTB13:
    .p2align 4,, 15
    .globl  main
    .type   main, @function
main:
.LFB34:
    .cfi_startproc
    flds    .LC4
    flds    .LC5
    flds    .LC6
    fld1
    flds    .LC8
    jmp     .L3
    .p2align 4,, 10
    .p2align 3
.L11:
    fxch    %st(3)
    fxch    %st(2)
    fxch    %st(1)
.L3:
    fld     %st(0)
    fmul    %st(1), %st
    flds    .LC0
    fld     %st(2)
    fmul    %st(1), %st
    faddp   %st, %st(2)
    flds    .LC1
    fsubr   %st, %st(2)
    fxch    %st(7)
    fmulp   %st, %st(2)
    fldz
    fucomip %st(2), %st
    fstp    %st(1)
    fxch    %st(1)
    fcmovbe %st(3), %st
    fxch    %st(2)
    fcmovnbe    %st(3), %st
    fstp    %st(3)
    fldl    .LC12
    fxch    %st(4)
    fucomip %st(4), %st
    fstp    %st(3)
    jb      .L10
    fld     %st(0)
    fadd    %st(2), %st
    fmuls   .LC9
    fld     %st(0)
    fmul    %st(1), %st
    fxch    %st(4)
    fmul    %st(1), %st
    faddp   %st, %st(4)
    fxch    %st(3)
    fsubp   %st, %st(4)
    fld     %st(3)
    fabs
    fldl    .LC12
    fucomip %st(1), %st
    jbe     .L11
    fstp    %st(0)
    fstp    %st(0)
    fstp    %st(0)
    fstp    %st(0)
    fstp    %st(0)
    xorl    %eax, %eax
    ret
.L10:
    fstp    %st(0)
    fstp    %st(0)
    fstp    %st(0)
    fstp    %st(0)
    movl    $1, %eax
    ret
    .cfi_endproc
.LFE34:
    .size   main, .-main
    .section    .text.unlikely
.LCOLDE13:
    .section    .text.startup
.LHOTE13:
    .section    .rodata.cst4,"aM",@progbits,4
    .align 4
.LC0:
    .long   1077936128
    .align 4
.LC1:
    .long   1071644672
    .align 4
.LC4:
    .long   3222194776
    .align 4
.LC5:
    .long   1074711128
    .align 4
.LC6:
    .long   3190690940
    .align 4
.LC8:
    .long   3215688991
    .align 4
.LC9:
    .long   1056964608
    .section    .rodata.cst8,"aM",@progbits,8
    .align 8
.LC12:
    .long   1073741824
    .long   1065646817
    .ident  "GCC: (GNU) 5.3.1 20151207
            (Red Hat 5.3.1-2)"
    .section    .note.GNU-stack,"",@progbits
```

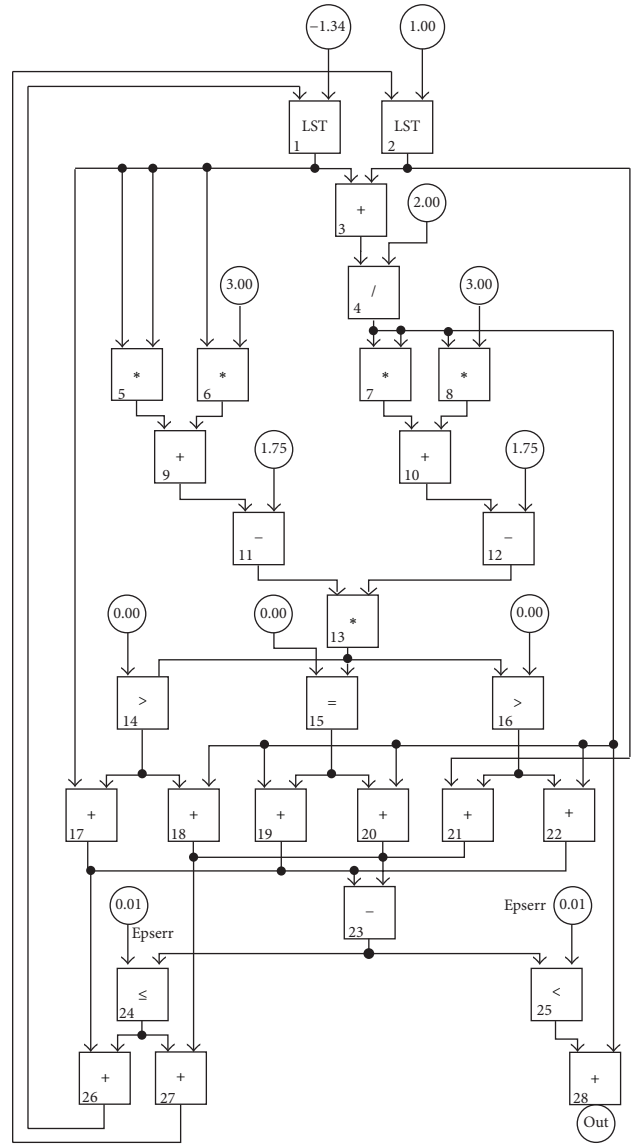
Box 3: Function root-finding with the bisection method: assembly code for a Pentium Dual CPU.

line (Section 2) compared with the x86 code. Another point is that we can easily evaluate the time required to execute a context without running it. Then, loaded the D# code, the accelerator works it out asynchronously while partial results flow between DFUs until the final result is not ready. Because the computation gets along without storing any partial data, another important fact of this organization is that both temporary data and instructions are eliminated as traffic over the memory access busses. Consequently, avoiding the communication traffic over the accelerator, the DFSC processor provides the advantage to naturally augment speedup and reduce latency drastically for a given technology.

6.2. Matrix Multiplication. Matrix multiplication $A(n,n) \times B(n,n)$ is an example of an intrinsically acyclic algorithm and constitutes the kernel for many linear algebra-based applications. Despite its algorithmic simplicity, it is computationally complex and memory intensive, $O(n^3)$ for two matrices $n \times n$. Moreover, its algorithm perfectly matches an interconnection mode configurable architecture since each dot product forms

Actor #	Func.	Left input	Right input	Output
1	LST	26	%–1.34	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

(a) D# code



(b) DPG representation

Figure 10: Function root-finding with the bisection method.

a reversed binary-tree graph. More in general, the DFSC processor is capable of executing any program represented by a DPG; however, for the sake of a more effective illustration, we prefer to focus on this simple example in this paper. For the matrix multiplication we compared three different execution architectures with the DFSC processor: the SIMD extension of an x86 architecture (shortly SIMD-x86), the tile-based FPGA architecture (shortly FPGA-tile-based) proposed by Campbell and Khatri [52], and the Cyclops-64 chip [53].

6.2.1. The Matrix Multiplication Evaluation Model. Given two matrices $A(N, M)$ and $B(M, P)$, where $a_{i,j}$ with $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, M$ is an element of A and $b_{j,k}$ with $j = 1, 2, \ldots, M$ and $k = 1, 2, \ldots, P$ is an element of B, the product $C(N, P) = A(N, M) \times B(M, P)$ can be expressed with $n_{\text{dp}} = N \times P$ independent dot products $(\underline{\mathbf{a}}_i \cdot \underline{\mathbf{b}}_k)$ of the N row vectors of A and P column vectors of B whose dimensions are M:

$$\begin{aligned} \underline{\mathbf{a}}_i &= \{a_{i1}, a_{i2}, \ldots, a_{iM}\} \quad i = 1, 2, \ldots, N, \\ \underline{\mathbf{b}}_k &= \{b_{1k}, b_{2k}, \ldots, b_{Mk}\} \quad k = 1, 2, \ldots, P. \end{aligned} \tag{5}$$

Figure 11(a) shows the DPG of the dot product in D# graphical representation where LI, RI, and O represent the left and right inputs, and the output sets of tokens, respectively, as defined in D# language. It shapes a reversed binary-tree graph with $2M - 1$ actors; M are organized in one level π of parallel multiplications and $M - 1$ are organized in $l = \lceil \log_2 M \rceil$ levels

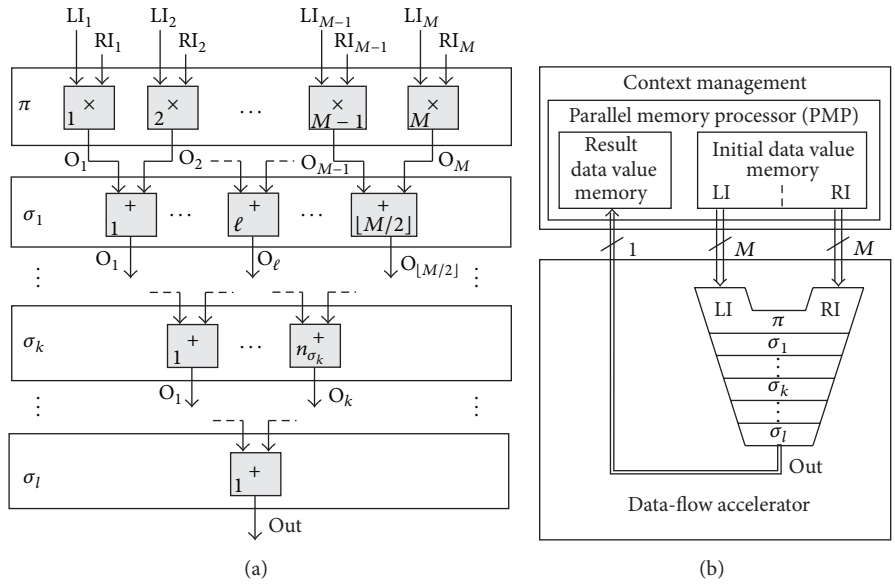


FIGURE 11: Dot product: (a) the DPG and (b) the generalized DFSC.

of σ additions, where each level $l_k \in [1 \le k \le l]$ broadens a number n_{σ_k} of parallel additions:

$$n_{\sigma_k} = \left\lfloor \frac{M - \sum_{j=1}^{k-1} n_{\sigma_j}}{2} \right\rfloor. \tag{6}$$

Figure 11(b) shows the *generalized* DFSC (shortly *g*DFSC) processor (in this context, the term generalized DFSC processor refers to an abstract DFSC processor architecture whose resources are always sufficient to compute any one dot product) with the related context management and data-flow accelerator. Like actors in the DPG, DFUs in the accelerator are organized in the same number of levels (stages); in the context management, the parallel memory processor (PMP) is organized in two stages—one to send the initial values and one to receive the final value. The latency parameters, which characterize a computation to the inside of the *g*DFSC processor, can be defined as follows.

Definitions. Let $\{\tau_{\mathrm{tr}_i} : 1 \le i \le M\}$ be the set of latencies required to transfer the corresponding single tokens between the PMP and accelerator registers; let $\{\tau_{m_i} : 1 \le i \le M\}$ be the set of latencies that each DFU requires for a multiplication operation; and let $\{\tau_{a_i} : 1 \le i \le M-1\}$ be the set of latencies that each DFU requires for an addition operation. The latency for the token transfer between the PMP and the accelerator is defined as $\tau_{\mathrm{tr}} = \max_i \tau_{\mathrm{tr}_i}$, the latency for the multiplication is defined as $\tau_m = \max_i \tau_{m_i}$, and the latency for the addition is defined as $\tau_a = \max_i \tau_{a_i}$.

It follows that, for a given M, the *g*DFSC computes the dot product in a time t_{dp}:

$$t_{\mathrm{dp}}(M) = \tau_{\mathrm{tr}} + \tau_m + l(M) \times \tau_a. \tag{7}$$

We point out that, when $2^r < M \le 2^{r+1}\ \forall r \in \mathbb{Z}^+$, t_{dp} remains constant although the number of operations changes. This occurs because, in the DPG, the number of levels does not change. Moreover, the relation between $t_{\mathrm{dp}}(M)$ and M can be expressed with the following step function:

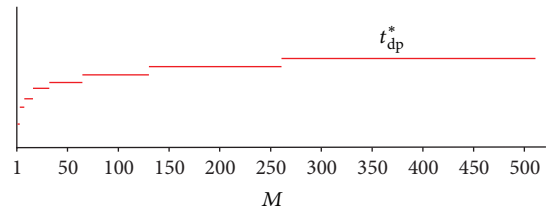


FIGURE 12: Computation time for the dot product $t^*_{\mathrm{dp}}(M)$.

$$t^*_{\mathrm{dp}}(M) = \sum_{i=1}^{n} t^i_{\mathrm{dp}}(M), \tag{8}$$

where t^i_{dp} is the dot product time when M varies in the ith domain $(2^{r^i}, 2^{(r+1)^i}]$.

Table 2 summarizes the features of the generalized processor configured to execute a dot product. Figure 12 draws the computation time $t^*_{\mathrm{dp}}(M)$ for different values of M, whereas *g*DFSC throughput rate TP is

$$\mathrm{TP} = \frac{1}{t^*_{\mathrm{dp}}(M)}, \tag{9}$$

while *g*DFSC computes the matrix product in a time $T_{\mathrm{mp}}(M)$:

$$T_{\mathrm{mp}}(M) = n_{\mathrm{dp}} \times t_{\mathrm{dp}}(M) \quad \text{with } n_{\mathrm{dp}} = N \times P. \tag{10}$$

To reduce T_{mp}, it is possible to apply the linear pipelining technique to the dot product computation because its DPG is naturally organized to support such a technique. In this case,

Table 2: Characteristics of the *g*DFSC processor for a dot product.

Level	DAC number	Operation type	time	Parallelism degree (spatial)
1	M	multiply	τ_m	M
2	$\lfloor M/2 \rfloor$	add	τ_a	$\lfloor M/2 \rfloor$
⋮	⋮	⋮	⋮	⋮
i	$\lfloor (M - \sum_{j=1}^{i-1} n_{\sigma_j})/2 \rfloor$	add	τ_a	$\lfloor (M - \sum_{j=1}^{i-1} n_{\sigma_j})/2 \rfloor$
⋮	⋮	⋮	⋮	⋮
$L = 1 + l$	1	add	τ_a	1
Computing time $t_{\text{dp}} = \tau_m + l \times \tau_a$				

the *h*HLDS firing rules can guarantee the determinate computation during the asynchronous execution. It is possible to further decrease T_{mp} using pipelined DFUs as well. Anyway, here we are only interested in showing *g*DFSC adaptability to simultaneously support different forms of parallelism and speed up an algorithm execution.

The dot product execution is composed of three sequential tasks—tsk_{tr} for the token transfer between PMP and accelerator, tsk_m for the token multiplication, and tsk_a for the token addition, whereas in the pipelined mode, the pipeline period τ_p is $\tau_p = \max(\tau_{\text{tr}}, \tau_m, \tau_a)$ and the pipeline throughput rate TP^P is

$$\text{TP}^P = \frac{1}{\tau_p}. \tag{11}$$

When $M \neq 2^i$, the pipelined execution needs interstage latches to correctly compute the dot product. In this case, the interstage latches are turned into adequate delays inside the initial data value memory of PMP. After that, the computation advances asynchronously.

Speedup. To fill all of the pipeline stages and produce the first result, the *g*DFSC takes a time $t_{\text{dp}}(M)$. After that, each dot product result comes out after every τ_p. Hence the total time T^P_{mp}, to process all of n_{dp} dot products in the $2 + l(M)$ stages pipeline, is

$$T^P_{\text{mp}}(M) = \left[(1 + l(M)) + n_{\text{dp}}\right] \times \tau_p, \tag{12}$$

and the speedup $\text{SP}(M)$ is given by

$$\text{SP}(M) = \lim_{n_{\text{dp}} \to \infty} \frac{T_{\text{mp}}(M)}{T^P_{\text{mp}}(M)} = 2 + l(M). \tag{13}$$

Anyway, for $n_{\text{dp}} \gg 2 + l(M)$ ($\gg$ refers to the wanted precision), we can assume $\text{SP}(M) = 2 + l(M)$. Besides, the number of flops n_{flop} in pipelined mode is given by

$$n_{\text{flop}}(M) = \frac{2M - 1}{\tau_p}. \tag{14}$$

6.3. DFSC Characterization. To characterize the DFSC, we used a custom board demonstrator with two Altera APEX20K1500E devices with 51840 Logic Elements (LEs) and 442368 RAM bits (without reducing available logic) inside the 216 Embedded System Blocks (ESBs) that allow the implementation of multiple memory functions (dual-port RAM, FIFO, ROM, etc.). The two devices are connected via six 132-bit buses to exploit the 808 tristate I/O user pins, and on the board is a pipelined SRAM memory to store contexts and tokens in case their number exceeds the local memory capacities of the management module. Finally, the board uses a PCI/interface to connect to the host. Our implemented instance consists of $n_d = 64$ DFUs (Section 4) and executes operations on 32-fixed-point operands. 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). Table 3 reports the resources required for Device-1 and Device-2 implementations. Please note that in a previous paper [17] we evaluated the latencies of a DFU, register-to-register, and the context switch which are 32 ns, 7 ns (device-to-device registers) and 4 ns (internal registers), and 32 ns, respectively. The matrix multiplication $C(n,n) = A(n,n) \times B(n,n)$ in D# consists of n^2 independent inner products (IPs) whose DPGs are organized in identically reversed binary-trees, each consisting of n multiplications and $n-1$ additions sequenced in $\log_2 + 1$ levels. Thanks to its shape, the inner product DPG is well suited for a naturally pipelined execution, thus allowing for speedup of the matrix multiplication computation.

Here we evaluate the product of matrices for A and B in pipelining. For the test we used matrix dimensions $n = 32$ and $n = 64$, respectively, with all matrix elements residing in the local data memory of the context management module. For $n = 32$ the inner product (IP) DPG is wholly mapped onto the DFSC processor, by means of 63 out of 64 DFUs available (Figure 13(a)). We point out that 64 is the biggest tile size that can be considered because the PMP does not need any optimization of load and store activities. For $n = 64$ we split the DPG into two sub-DPGs as we did for $n = 32$. Then, we execute the two inner products $\text{IP}' = n_{a'_{i/2}} \times n_{b'_{j/2}}$ and $\text{IP}'' = n_{a''_{i/2}} \times n_{b''_{j/2}}$ and use the DFU #64 to add the two results as shown in Figure 13(b). We would like to point out that this decomposition does not hinder the execution (throughput) of an inner product IP when pipelining occurs because the DPG in Figure 13(b) behaves as if all the required DFUs were in the DFSC but doubles the number of IPs to execute.

Table 3: DFSC: APEX 20K-1500E FPGA resources (total LEs = 51840; total RAM bits = 442368).

	Context management (Device-1)				
	I/O Token buffers	32 DFUs	CCM	PMP	Scheduler
LEs	0	11766	187	384	215
RAM bits	6336	0	8448	406912	0
	Accelerator (Device-2)				
	I/O Token buffers	32 DFUs	Switch interconnect	DF-Code memory	
LEs	0	11766	36365	386	
RAM bits	6336	0	0	1408	

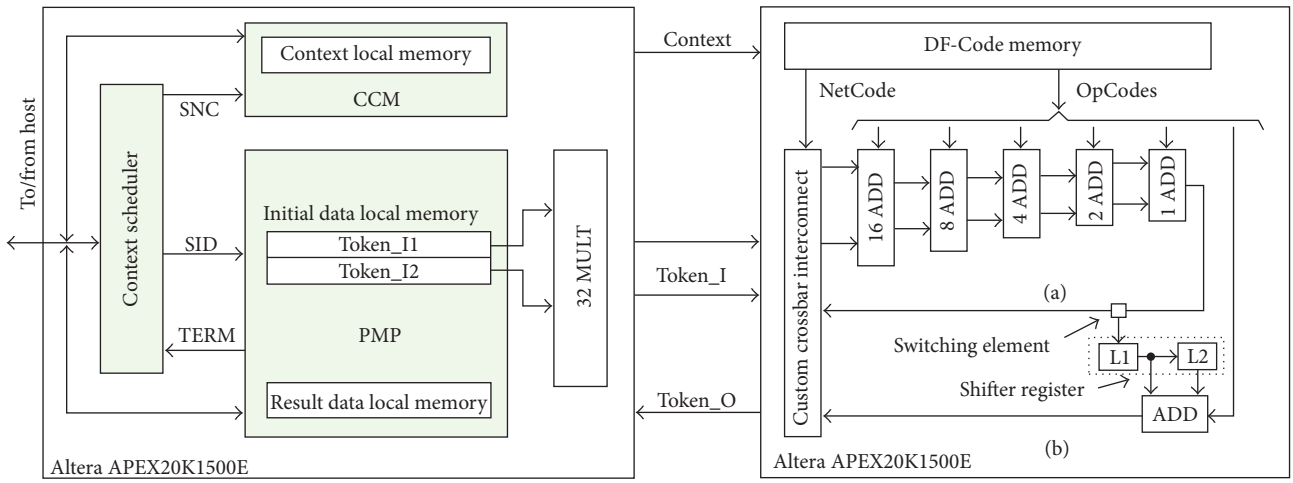


Figure 13: DPG for the inner products (a) $n = 32$ and (b) $n = 64$.

6.4. SIMD-x86 and FPGA-Tile-Based versus DFSC. To make a fair DFSC comparison between the SIMD-x86 and the FPGA-tile-based [52], which act at different clock rates (2 GHz for the SIMD-x86 and 400 MHz for the FPGA-tile-based) and are based on different state-of-the-art technologies, we measured, for the matrix multiplication, the performance in terms of cycles per instructions rather than in terms of GFLOPs because, in [52], authors used this parameter to evaluate their FPGA-tile-based processing element. To avoid the large time penalty incurring each time to fetch an element from x86 L2 cache and to compare our results with those reported for the FPGA-tile-based, we used matrices of size $n = 32$ and $n = 64$, respectively. For the DFSC we determined the number of stages and the stage clock rate involved in an IP computation in pipelining as well.

DFSC Execution. The parallel execution of the 32 multiplications on the management module requires the move from the PMP local memory to the 32 DFU of $2 \times 32 = 64$ tokens (In1 and In2). Since the token is 33 bits, the total number of bits to be moved to the 32 DFUs is $33 \times 64 = 2112$. By exploiting the internal FPGA interconnect, we can transfer 2×4 tokens (264+264 bits) at a time. In total this takes 8x internal register-to-register transfers and has a latency equal to $8 \times 4 = 32$ ns, while the multiplication needs 30 ns. Transferring 32×33-bit tokens of the product from the management module to the accelerator requires 4x external register-to-register transfers and latency of $4 \times 7 = 28$ ns, whereas inside the accelerator the transfer of the input buffers to the 32 DFUs in the first level 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 it with a stage latency $\tau_p = 32$ ns (clock rate 31,2 MHz). 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 Figure 13(b) produces c_{ij}. In this case we add the first latch latency (4 ns) to the DFU #63 latency while the second latch latency is added to the DFU #64 latency. Consequently, we have $\tau_p = 34$ ns (clock rate 29,4 MHz), $n_l = \log_2(64) + 1 = 7$ 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 $n_c = 2 \times 64^2 + 13 = 8205$.

SIMD-x86 Execution. Multimedia Extension (MMX) technology provides acceleration through SIMD parallelism providing SIMD multiplication and addition instructions where two 32-bit integer values are operated on at once. To

Table 4: SIMD-x86 and FPGA-tile-based versus DFSC.

n	SIMD-x86 cycles	FPGA-tile-based cycles	DFSC cycles
32	16131	2081	1035
64	130098	16417	8205

determine the number of clock cycles, we used the related built-in hardware performance counter.

FPGA-Tile-Based Execution. The FPGA-tile-based considered [52] implements the matrix multiplication algorithm in a Xilinx Virtex-4 XC4VSX55-12 FPGA (which is similar to the one we used for the DFSC) and consists of an array of PEs as in a CGRA. Each PE operates at 400 MHz independently of the other PEs in the array and the operating frequency of the PE array is independent of matrix dimension. The PE structure consists of one input each from matrices A and B, a multiplier accumulator (MAC), and a result FIFO. The inputs from matrices A and B, one word each per clock cycle, are implemented using dedicated routes from the BlockRam memory associated with the multiplier, thus eliminating the routing and resource delay penalty.

Each matrix is partitioned into m BlockRam banks. Each of the banks dedicated to A stores $k = (n/m)$ words of each column in A, for every row of A. Each of the banks dedicated to B stores $k = (n/m)$ words of each row in B, for every column of B, requiring a number of cycles of 2081 for $n = 32$ and 16417 for $n = 64$.

Comparison Results. Table 4 shows the comparison results in terms of number of cycles and execution time for matrix multiplications with $n = 32$ and $n = 64$, respectively.

As we can observe, regarding the number of cycles the DFCS processor performs better than a SIMD-x86 (roughly 15 times for $n = 32$ and 8 times for $n = 64$) and the FPGA-tile-based (roughly 2 times for $n = 32$), but regarding the execution time it performs worse than them. However, if we consider that the technology of current FPGAs can allow DFUs to operate at 400 MHz, the Data-Flow Soft-Core becomes quite interesting and competitive to a dedicated out-of-order processor or dedicated solutions on FPGAs.

6.5. Cyclops-64 versus DFSC. For this comparison, we interpolated the data assuming an FPGA with 300 floating-point DFUs and a clock rate at 500 MHz. From the other part, the multithreading many-core platform IBM Cyclops-64 [53] consists of 160 cores on a single chip. In particular, each Cyclops-64 processor (C64 for short) consists of an 80 cores with two Thread Units (TU) per core, a port to the on-chip interconnect, external DRAM, and a small amount of external interface logic. The TU is a simple 64-bit, single issue, in-order RISC processor operating at a moderate clock rate (500 MHz). A software thread maps directly onto a TU and the execution is nonpreemptive; that is, the microkernel will not interrupt the execution of a user application unless an exception occurs (no context switch). Each thread controls a region of the scratchpad memory, allocated at boot time.

Table 5: Cyclops-64 versus simulated 300 DFU DFSC-based processor.

	Machine characteristics	
	Cyclops-64 node	DFSC
Number of cores	160 (classical)	300 (DFUs)
Memory hierarchy level	3	1
Architecture model	Hybrid	Pure data-flow
Program execution model	Tiny-Thread (TNT)	Interconnected DPGs
Performance	Simulated	Interpolated
	70.0 GFlops[1]	123 GFlops[2]
	0.43 GFlops	0.41 Gflops

[1]Input data on a chip. [2]Included transfer time from PMP to accelerator.

Moreover, there is no hardware virtual memory manager so the three-level memory hierarchy is visible to the programmer. The comparison with the Cyclops-64 is interesting in this context because it uses a data-flow-based execution model. For the C64 platform we refer to the run with a FAST Simulator [54] of a test based on a highly optimized dense matrix multiplication using both on-chip and off-chip memory. Both static and dynamic scheduling strategies were implemented [55–57]. We optimized the code so that the DFSC execution could take place in pipeline. In our case, we first performed the multiplications in parallel and then we performed the additions in $\log_2(n)$ stages.

This evaluation compares the GFLOPS-per-core of the two architectures when the two processors execute a matrix multiply 300×300. The result of this comparison is reported in Table 5.

7. CGRA versus DFSC

In this section we discuss and compare some CGRA architecture with the DFSC processor *MorphoSys System* [46] having a MIPS-like Tiny RISC processor with extended instruction set that executes sequential tasks, a mesh-connected 8 by 8 reconfigurable array (RA), a frame buffer for intermediate data, context memory, and DMA controller. The RA is divided into four quadrants of 4 by 4 16-bit RCs each, featuring ALU, multiplier, shifter, register file, and a 32-bit context register for storing the configuration word. The interconnect network uses 2D mesh and a hierarchical bus to span the whole array. Tiny RISC extra DMA instructions initiate data transfers between the main memory and the "frame buffer" internal data memory for blocks of intermediate results, 128 by 16 bytes in total. Programming frameworks for RAs are highly dependent on structure and granularity and differ by language level. For MorphoSys, it is assembler level. It is supported by a SUIF-based compiler for host and development tools for RA. Configuration code is generated via a graphical user interface or manually from an assembler level source also usable to simulate the architecture from VHDL. In contrast, the DFSC processor

exhibits MIMD functionality, its interconnect is crossbar-like so that intermediate results are asynchronously passed directly from a data-flow functional unit to another avoiding so to store partial results, and its context switching is managed by the Context Configuration Manager (CCM) according to the operations to be executed. As an example, in pipelined operations involving data-flow functional units as pipeline stages, the context does not change while a new input data stream arrives.

ADRES [58], designed for Software Defined Radio applications, is a reconfigurable processor with tightly coupled VLIW processor. Each reconfigurable cell has mainly a functional unit (FU) and a register file that contains a 32-bit ALU which can be configured to implement one of several functions including addition, multiplication, and logic functions, with two small register files. It utilizes MorphoSys communication mechanism and resolves resource conflict at compile time using modulo scheduling. If the application requires functions which match the capabilities of the ALU, these functions can be very efficiently implemented in this architecture. Applications, written in ANSI-C, are transformed into an optimized binary file. However, exploiting instruction level parallelism (ILP) out of ADRES architecture can cause hardware and software (compiler) inefficiency because of the heavily ported global register file and multi-degree point-to-point connections. In contrast, in the DFSC, since the accelerator does not store intermediate data during an execution, the execution of data-dependent loops (cycles) happens asynchronously with the only control of the data-flow functional unit (DFU) firing rule implementation.

KressArray [59] is a 2D mesh of rDPUs (reconfigurable datapath units) physically connected through local nearest neighbor (NN) link-spots and global interconnection. The KressArray is a supersystolic array. Its interconnect fabric distinguishes 3 physical levels: multiple unidirectional and/or bidirectional NN link-spots, full length or segmented column or row back buses, and a single global bus reaching all rDPUs (also for configuration). Each rDPU can serve for routing only, as an operator, or an operator with extra routing paths. With the new Xplorer environment [60] rDPUs also support other operators (branching, while and do-while loops, etc.). Differently, the DFSC processor supports asynchronous computations without the need for global synchronization, uses a crossbar-like interconnect to link the DFUs, and allows on-the-fly context switching.

8. Conclusions

This paper presented the Data-Flow Soft-Core architecture. We evaluated this new concept of soft-core by using two simple test cases based on the bisection method and matrix multiply programs. The results that we obtained show a sizable reduction of the number of microoperations, typical in conventional core, and a competitive advantage both against reduced data-flow engines like the classical out-of-order processors and dedicated FPGAs and against the data-flow-based Cyclops-64. We believe that this processor needs further development but represents a first step toward a more flexible execution of generic programs on a scalable reconfigurable platform. The Data-Flow Soft-Core introduces a new level of programmability that enhances the usability of FPGA platforms through the use of data-flow instructions rather than pretending to convert control-flow instructions, like what happens with the bisection method and matrix multiplication algorithms.

```
input (a, b = 1, c)
  repeat
    if a > 1 then a ≔ a \ 2
    else a ≔ a ∗ 5
    b ≔ b ∗ 3;
  until b = c;
  d ≔ a;
output (d)
```

Listing 1: A sample program.

Appendix

A. Classical Model versus *h*HLDS

To better understand the difference between the classical model and *h*HLDS, let us consider the sample program pseudocode in Listing 1, where a, b, and c are the input and d is the output data. Figure 14 shows the equivalent well-behaved DPGs with the classical model and *h*HLDS, respectively. For each DFG, the part inside the grey rectangle computes something, while the remaining part checks the end of the computation and outputs d. The DPG in Figure 14(a) has five types of actors: α for merge, β for switch, γ for gate, δ for decider, and ϵ for operator. Each of them has heterogeneous I/O conditions, link-spots (places to hold tokens), and tokens; data link-spots hold data tokens and control link-spots hold control tokens. As can be observed, this representation presents some issues. First, to comprehend how the DPG works, we have to follow the flow of two types of tokens along a graph where there are actors with different numbers of input and output link-spots that can consume and produce them. Then, the initial behavior of the actors α, β, and γ depends on their position in the DPG rather than program input values. The initial control tokens F and T (red dots in Figure 14(a)) for the α and β actors are automatically present on their control arcs but have different values (false and true, resp.) although they share the same link-spot. As this condition cannot be true, it can be only managed via software because the classical DPG represents the schema of what we want to do, not what we really do. However, these control values, even if they might be deduced, are not a program input but a programmer's trick to allow the computation to start correctly. Besides, not all functions associated with actors are defined in the same domain and assume a value in the same codomain. For example, if $\mathbf{R}$ is a subset of real numbers, $\mathbb{B}$ is the set of boolean values, and $\mathbf{W}$ is the set $\mathbf{R} \times B$, we note that the

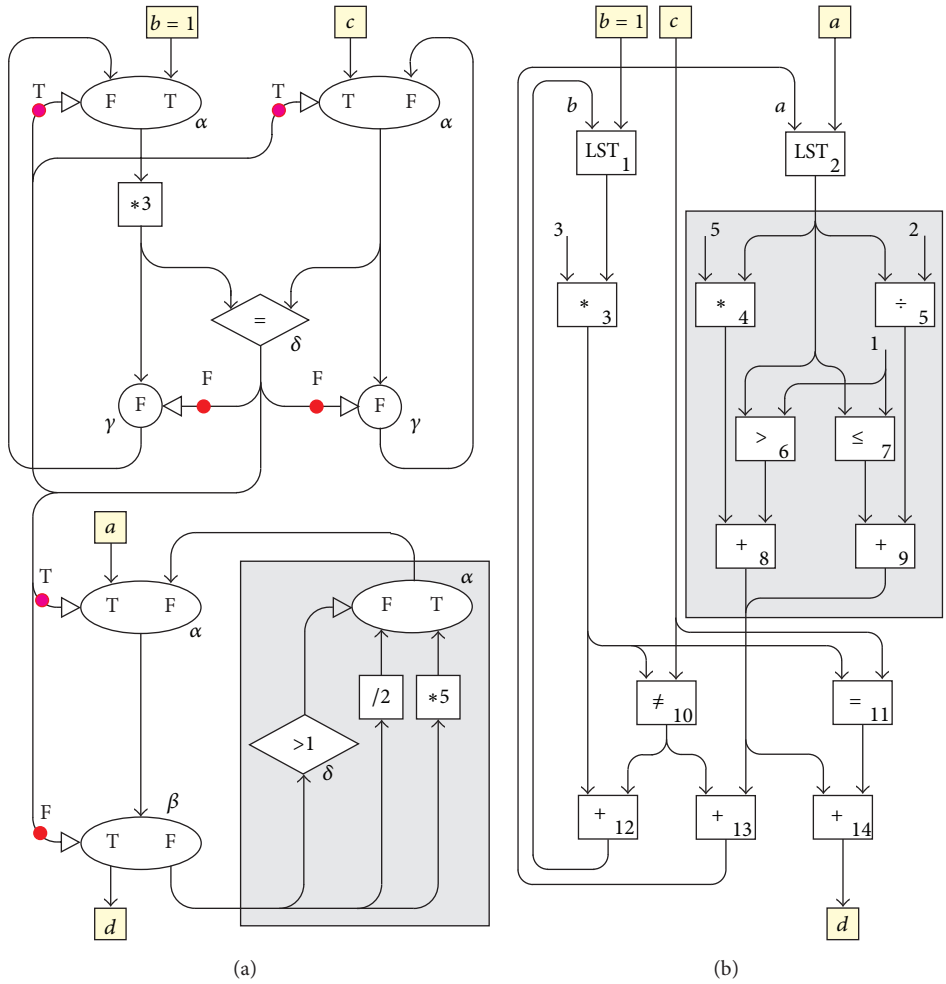


FIGURE 14: Data-flow program graphs for the classical model (a) and *h*HLDS (b).

functions associated with arithmetical actors are defined and assume values in **R**, the function associated with the actor δ is defined in **R** but assumes values in $\mathbb{B}$, and the functions associated with actors α, β, and γ are defined in $\mathbf{R} \times \mathbb{B}$ but assume values in **R**. We remark that, in the classical static-dataflow model, the firing rule imposes that an actor fires only if its input tokens are present and its output is empty. As a result of this, the asynchronous computation of a DPG requires a handshake communication between the producer and consumer actors, which (handshaking) increases the communication overhead and complicates the control signal management. Differently, the DPG in Figure 14(b) is formed by actors with only homogeneous I/O conditions; its initial behavior only depends on the program input values; actors consume and produce only data tokens; the validity of a token is the only in charge of the well-behaved execution. Furthermore, the homogeneity of the actor I/O conditions and tokens constitutes the determining factor to create the one-to-one mapping between actors of *h*HLDS and DFUs of a truly asynchronous data-flow accelerator that executes directly in hardware data-flow program graphs.

B. Chiara Language

To explain how the matrix multiplication function (program) works in Chiara, first we briefly describe the language features and then we comment on the program.

Chiara language has been explicitly designed to program in FP style [31] and obtain the program code in D#. Like FP, the Chiara programming system is a tuple:

$$(O, F, \mathscr{F}, :, D), \tag{B.1}$$

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

Objects include atoms, sequences, and the *undefined* special object $\perp$, called *bottom*, which is used mostly to denote errors. Atoms include integer fixed and floating-point numbers, true values, characters, and strings. Sequences are denoted with angle brackets. Therefore the three objects

$$1 \quad \langle 1, 2, 3 \rangle \quad \langle \langle 1, 2 \rangle \langle 3, 4 \rangle \rangle \tag{B.2}$$

represent valid Chiara objects.

Chiara includes two kinds of operators that can be applied to objects: *elementary* and *combinator* operators. Up to now elementary operators are those included in Table 1. Combinator operators represent operators that affect the structure of the objects on which they are applied. As an example, there are combinators that extract objects out of a sequence, combine sequences, transpose sequences of sequences, and so forth. Functional forms, augmented with new ones as *case*, *repeat*, and the binary insert "!," are used to combine existing elementary functions and combinators to create new functions. The application ":" is the operation that denotes the object which is the result of applying the operator to an object (e.g., $+ : \langle 1, 2\rangle = 3$).

All functions f in F map objects into objects and are *bottom-preserving*: $f : \perp = \perp$, for all f in F. Every function in F can be a combinator (i.e., a built-in function), a user-defined function, or a functional form. If the computation for $f : x$ yields the object $\perp$, we say f is undefined at x; that is, f has no meaningful value at x. This is what happens when the relational actor $\mathscr{R}$'s predicate is not satisfied (Section 2.2).

Finally, a definition is an expression that assigns a name to a function, def *name* = f, where *name* in an unused symbol. The set of definitions def $\in$ D is *well formed* if no two left sides are the same. For example, the expression

$$\text{def max} = \text{ge} \circ [1, 2] \longrightarrow 1; 2 \tag{B.3}$$

is well formed and defines max as the function that evaluates the maximum between the first two objects in a sequence. The program has two steps: (ge) $\circ$ ($[1, 2] \rightarrow 1; 2$) reading from right to left, and each is applied in turn to the result of its predecessor.

ge is the relational operator "*greater than or equal to*"; $\circ$ is the functional *composition* given any functions f and g and an object x, $f \circ g : x \equiv f : (g : x)$; [ ] is the functional *construction* that applies the list of functions inside the square brackets to an object x, $[f_1, \ldots, f_n] : \langle x\rangle \equiv \langle f_1 : x, \ldots, f_n : x\rangle$; 1 and 2 integers are two of the combinator *selector*, $i : \langle x_1, \ldots, x_i, \ldots, x_n\rangle = x_i$; $\rightarrow$ is the functional *condition* identifier; $1; 2$ is the selection to apply to the two objects $\langle x_1, x_2\rangle$. When max : $\langle x_1, x_2\rangle$, if ge : $\langle x_1, x_2\rangle = T$, then $1 : \langle x_1, x_2\rangle$ and max = x_1; otherwise $2 : \langle x_1, x_2\rangle$ and max = x_2.

Let us now comment on the Chiara program code shown in Box 1.

We use // to comment on a line of code; DP is the user-defined function for the dot product; the penultimate row is the program (like a main) function, which executes the matrix multiplication; stop ends the program. The program has four steps: (&& DP) $\circ$ (& distl) $\circ$ distr $\circ[1, \text{trans} \circ 2]$, beginning with $[1, \text{trans} \circ 2]$. It allows the multiplication of any pair of conformable matrices $A(m, k)$ and $B(k, n)$ as argument, represented as sequences of their m rows $\langle a_1, \ldots, a_m\rangle$ and k rows $\langle b_1, \ldots, b_k\rangle$ for A and B, respectively.

&, $\circ$, and [ ] are the functional forms: *apply to all*, *composition*, and *construction*, respectively, while *trans* and ! are the combinator operators transpose and binary insert, respectively. trans, applied to two equal-length sequences, produces their transpose; !, applied to a function with argument a sequence, produces a reversed binary-tree where each actor clones an operator, and each starting link-spot holds an element of the sequence as shown in Figure 9.

The first step produces $\langle A, B^T\rangle$ by means of the combinator *trans*, which transposes B. The second step produces the sequence $\langle\langle a_1, B^T\rangle, \ldots, \langle a_m, B^T\rangle\rangle$ applying the combinator *distr* (distribute from right), distr : $\langle\langle x_1, \ldots, x_q\rangle, y\rangle = \langle\langle x_1, y\rangle, \ldots, \langle x_q, y\rangle\rangle$.

The third step produces the sequence of row and column pairs. & distl first uses the functional & (apply to all) to the combinator *distl* (distribute from left) which yields distl : $\langle\langle a_1, B^T\rangle, \ldots,$ distl : $\langle a_m, B^T\rangle\rangle$ and then uses distl, which yields $\langle\langle\langle a_1, b_1^T\rangle, \ldots, \langle a_1, b_n^T\rangle\rangle, \ldots, \langle\langle a_m, b_1^T\rangle, \ldots, \langle a_m, b_n^T\rangle\rangle\rangle$.

In the fourth step, the functional &&DP, or &(&DP), causes &DP to be applied to each $\gamma_i = \langle\langle a_i, b_1^T\rangle, \ldots, \langle a_i, b_n^T\rangle\rangle$, which in turn causes DP to be applied to each row and column pair in each γ_i. In the function DP, as defined in the code (Box 1), ! is the functional *binary insert*. Prefixed a function f applied to a sequence $\text{sq}_r = \langle x_1, \ldots, x_r\rangle$ with it, ! distributes f like a reversed binary-tree of $r-1$ identical functions f. The first level of the tree consists of $r/2$ functions whose inputs are the pairs of the sequence $\langle\langle x_1, x_2\rangle, \ldots, \langle x_{r-1}, x_r\rangle\rangle$ if r is even and the pairs of the sequence $\langle\langle x_1, x_2\rangle, \ldots, \langle x_{r-2}, x_{r-1}\rangle, x_r\rangle$ if r is odd; $!f : \langle \text{sq}_r\rangle = f : \langle !f : \langle x_1, \ldots, x_p\rangle, !f : \langle x_{p+1}, \ldots, x_r\rangle\rangle$, where $p = 2^{\lfloor \log_2 r\rfloor}$ if $p \neq r$; else $p = r/2$. The result of the last step is the sequence of rows comprising the product matrix. If A and B are not conformable, the result is $\perp$.

As it can be observed, a Chiara program is nothing but a set of function definitions plus an expression (i.e., a function applied to an object) that, once evaluated, will represent the result of the program.

As Chiara programs are variable-free, it can be easy to 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. This point is very important because it is possible to devise an initial data distribution over several DFSCs and then to follow the well-defined instructions present in the program code to execute the routing of the data in such a way that they reach the destinations where they have to be consumed. For example, the combinator trans moves data between places without performing any kind of actual computation but sequence-to-sequence or sequence-to-object transformations since it turns out to involve just a routing function. In our case, trans tells the compiler that the elements of the matrix B have to be organized by column and not by row.

Competing Interests

The authors declare that they have no competing interests.

References

[1] M. J. Flynn, O. Mencer, V. Milutinovic et al., "Viewpoint moving from petaflops to petadata," *Communications of the ACM*, vol. 56, no. 5, pp. 39–42, 2013.

[2] https://www.tensorflow.org/.

[3] M. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi, *Guide to DataFlow Supercomputing*, Springer, Berlin, Germany, 2015.

[4] S. Wesner, L. Schubert, R. M. Badia, A. Rubio, P. Paolucci, and R. Giorgi, "Special section on terascale computing," *Future Generation Computer Systems*, vol. 53, pp. 88–89, 2015.

[5] T. L. Sterling and M. J. MacDonald, "The realties of high performance computing and dataflow's role in it: lessons from the NASA HPCC program," in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (PACT '93)*, M. Cosnard, K. Ebcioglu, and J.-L. Gaudiot, Eds., pp. 165–176, North-Holland, Orlando, Fla, USA, January 1993.

[6] D. Culler, K. Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," in *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, 20–22 January, 1993*, M. Cosnard, K. Ebcioglu, and J.-L. Gaudiot, Eds., pp. 153–164, North-Holland, Orlando, Fla, USA, 1993.

[7] https://www.maxeler.com.

[8] O. Lindtjorn, R. G. Clapp, O. Pell, H. Fu, M. J. Flynn, and O. Mencer, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, 2011.

[9] O. Pell, J. Bower, R. Dimond, O. Mencer, and M. J. Flynn, "Finite-difference wave propagation modeling on special-purpose dataflow machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 5, pp. 906–915, 2013.

[10] S. Weston, J. Spooner, S. Racanière, and O. Mencer, "Rapid computation of value and risk for derivatives portfolios," *Concurrency Computation Practice and Experience*, vol. 24, no. 8, pp. 880–894, 2012.

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

[12] G. Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3–9, 2002.

[13] L. Verdoscia, "A dataflow machine architecture for static dataflow program graphs," *IPSI BgD Transactions on Advanced Research (TAR)*, vol. 12, no. 1, pp. 31–40, 2016.

[14] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing hardware accelerators in scientific applications: a case study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 58–68, 2011.

[15] G. Theodoridis, D. Soudris, and S. Vassiliadis, "A survey of coarse-grain reconfigurable architectures and cad tools," in *Fine-and Coarse-Grain Reconfigurable Computing*, S. Vassiliadis and D. Soudris, Eds., pp. 89–149, Springer, Amsterdam, Netherlands, 2007.

[16] L. Verdoscia, R. Vaccaro, and R. Giorgi, "A matrix multiplier case study for an evaluation of a configurable dataflow-machine," in *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*, ACM, Ischia, Italy, May 2015.

[17] L. Verdoscia, R. Vaccaro, and R. Giorgi, "A clockless computing system based on the static dataflow paradigm," in *Proceedings of the Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM '14)*, pp. 30–37, Edmonton, Canada, August 2014.

[18] A. Chattopadhyay, "Ingredients of adaptability: a survey of reconfigurable processors," *VLSI Design*, vol. 2013, Article ID 683615, 18 pages, 2013.

[19] D. Sheldon, R. Kumar, R. Lysecky, F. Vahid, and D. Tullsen, "Application-specific customization of parameterized FPGA soft-core processors," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '06)*, pp. 261–268, San Jose, Calif, USA, November 2006.

[20] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '01)*, pp. 564–570, ACM, Yokohama, Japan, 2001.

[21] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[22] B. De Sutter, P. Raghavan, and A. Lambrechts, "Coarse-grained reconfigurable array architectures," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., pp. 449–484, Springer, New York, NY, USA, 2010.

[23] J. Mikloško and V. E. Kotov, "Data flow computer architecture," in *Algorithms, Software and Hardware of Parallel Computers*, J. Mikloško and V. E. Kotov, Eds., pp. 323–358, Springer, New York, NY, USA, 1984.

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

[25] K. Gostelow and Arvind, "A computer capable of exchanging processing elements for time," Tech. Rep., Department of Information and Computer Science, University of California, 1976.

[26] J. Dennis, J. Fosseen, and J. Linderman, "Data flow schemas," in *International Symposium on Theoretical Programming*, A. Ershov and V. A. Nepomniaschy, Eds., vol. 5 of *Lecture Notes in Computer Science*, pp. 187–216, Springer, Berlin, Germany, 1974.

[27] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: a survey and a cost model," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 36:1–36:24, 2011.

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

[29] L. Verdoscia and R. Vaccaro, "Position paper: validity of the static dataflow approach for exascale computing challenges," in *Proceedings of the Data-Flow Execution Models for Extreme Scale Computing (DFM '13)*, pp. 38–41, IEEE, Edinburgh, Scotland, September 2013.

[30] L. Verdoscia, M. Danelutto, and R. Esposito, "CODACS prototype: CHIARA language and its compiler," in *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pp. 864–870, Tokyo, Japan, March 2004.

[31] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the Association for Computing Machinery*, vol. 21, no. 8, pp. 613–641, 1978.

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

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

[34] R. Giorgi, "TERAFLUX: exploiting dataflow parallelism in teradevices," in *Proceedings of the ACM Computing Frontiers Conference (CF '12)*, pp. 303–304, Cagliari, Italy, May 2012.

[35] M. Solinas, R. M. Badia, F. Bodin et al., "The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices," in *Proceedings of the Euromicro Conference on Digital System Design (DSD '13)*, pp. 272–279, IEEE, Los Alamitos, Calif, USA, September 2013.

[36] R. Giorgi, R. M. Badia, F. Bodin et al., "TERAFLUX: harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, vol. 38, no. 8, part B, pp. 976–990, 2014.

[37] P. Burgio, C. Alvarez, E. Ayguad et al., "Simulating next-generation cyber-physical computing platforms," *Ada User Journal*, vol. 36, no. 4, pp. 259–263, 2015.

[38] R. Giorgi and A. Scionti, "A scalable thread scheduling co-processor based on data-flow principles," *Future Generation Computer Systems*, vol. 53, pp. 100–108, 2015.

[39] R. Giorgi, "Exploring dataflow-based thread level parallelism in cyberphysical systems," in *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 1–6, Como, Italy, May 2016.

[40] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer, "Architectural support for fault tolerance in a teradevice dataflow system," *International Journal of Parallel Programming*, vol. 44, no. 2, pp. 208–232, 2016.

[41] R. Giorgi, "Transactional memory on a dataflow architecture for accelerating haskell," *WSEAS Transactions on Computers*, vol. 14, pp. 794–805, 2015.

[42] R. Giorgi and P. Faraboschi, "An introduction to DF-threads and their execution model," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW '14)*, pp. 60–65, IEEE, Paris, France, October 2014.

[43] D. Burger, S. W. Keckler, K. S. McKinley et al., "Scaling to the end of silicon with EDGE architectures," *Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[44] S. Swanson, A. Schwerin, M. Mercaldi et al., "The WaveScalar architecture," *ACM Transactions on Computer Systems*, vol. 25, no. 2, article 4, 2007.

[45] M. Murakawa, S. Yoshizawa, I. Kajitani et al., "The GRD chip: genetic reconfiguration of DSPs for neural network processing," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 628–639, 1999.

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

[47] G. Donohoe, T. Le, D. Buehler, and P.-S. Yeh, "Graphical design environment for a reconfigurable processor," in *Proceedings of the 7th International Conference Mil/Aerospace Applications of Programmable Logic Devices (MAPLD '04)*, Washington, DC, USA, September 2004.

[48] J. Teifel and R. Manohar, "An asynchronous dataflow FPGA architecture," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1376–1392, 2004.

[49] A. Takayama, Y. Shibata, K. Iwai, and H. Amano, "Dataflow partitioning and scheduling algorithms for wasmii, a virtual hardware," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, R. Hartenstein and H. Grnbacher, Eds., vol. 1896 of *Lecture Notes in Computer Science*, pp. 685–694, Springer, Berlin, Germany, 2000.

[50] S. Windh, X. Ma, R. J. Halstead et al., "High-level language tools for reconfigurable computing," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 390–408, 2015.

[51] G. Passas, M. Katevenis, and D. Pnevmatikatos, "A 128 × 128 × 24Gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6%of their area," in *Proceedings of the 4th ACM/IEEE International Symposium on Networks on Chip (NOCS '10)*, pp. 87–95, Grenoble, France, May 2010.

[52] S. J. Campbell and S. P. Khatri, "Resource and delay efficient matrix multiplication using newer FPGA devices," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06)*, pp. 308–311, Philadelphia, Pa, USA, April 2006.

[53] Y. P. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. Gao, "A study of the on-chip interconnection network for the IBM cyclops64 multi-core architecture," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, p. 10, April 2006.

[54] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Fast: a functionally accurate simulation toolset for the cyclops64 cellular architecture," in *Proceedings of the Workshop on Modeling, Benchmarking, and Simulation (MoBS '05), in Conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pp. 11–20, 2005.

[55] E. Garcia, I. Venetis, R. Khan, and G. Gao, "Optimized dense matrix multiplication on a many-core architecture," in *Euro-Par 2010—Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds., vol. 6272 of *Lecture Notes in Computer Science*, pp. 316–327, Springer, 2010.

[56] I. E. Venetis and G. R. Gao, "Mapping the LU decomposition on a many-core architecture: challenges and solutions," in *Proceedings of the 6th ACM Conference on Computing Frontiers (CF '09)*, pp. 71–80, ACM, Como, Italy, May 2009.

[57] E. Garcia, D. Orozco, R. Pavel, and G. Gao, "A discussion in favor of dynamic scheduling for regular applications in many-core architectures," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)*, pp. 1591–1600, Shanghai, China, May 2012.

[58] 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 Application*, P. Y. K. Cheung and G. A. Constantinides, Eds., vol. 2778 of *Lecture Notes in Computer Science*, pp. 61–70, Springer, Berlin, Germany, 2003.

[59] R. W. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of the Design Automation Conference, IFIP International Conference on Hardware Description Languages, and IFIP International Conference on Very Large Scal (ASP-DAC/CHDL/VLSI '95)*, pp. 479–484, Chiba, Japan, August-September 1995.

[60] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '00)*, pp. 163–168, Yokohama, Japan, June 2000.