A matrix multiplier case study for an evaluation of a configurable Dataflow-Machine

Lorenzo Verdoscia and Roberto Vaccaro
Institute for High Performance Computing and Networking - CNR
Via Castellino, 111 - 80131 Napoli, Italy
lorenzo.verdoscia@na.icar.cnr.it

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

ABSTRACT

Configurable computing has become a subject of a great deal of research given its potential to greatly accelerate a wide variety of applications that require high throughput. In this context, the dataflow approach is still promising to accelerate the kernel of applications in the field of HPC. That tanks to a computational dataflow engine able to execute dataflow program graphs directly in a custom hardware. On the other hand, evaluating radically different models of computation remains yet an open issue. In this paper we present as case study the matrix multiplication that constitutes the fundamental kernel of the linear algebra. The evaluation takes into account the execution of the matrix product both in non-pipelined and pipelined modes. Results obtained running the execution of the two modes on an FPGA-based demonstrator show the validity of the configurable Dataflow-Machine. Moreover, at the same throughput, the power consumption is expected to be lower than in clock-based systems.

Categories and Subject Descriptors

C.0 [**System Specification Methodology**]: Other Architecture Styles—*Data-flow architectures*; D.1 [**Programming Techniques**]: Functional Programming

General Terms

Design, Languages, Algorithms

Keywords

Functional languages, data-flow program graphs, many-cores, matrix multiplication

1. INTRODUCTION

Terms like *configurable* or *reconfigurable* computing are not new, and in the past they had a more general meaning than now [5]. In contrast, nowadays, FPGA-based configurable computing has become a subject of a great deal of research given its potential to greatly accelerate a wide variety of applications that require high throughput. Its key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. However, at the best of our knowledge, this flexibility still presents a high context switch time [12] when two different task processes (threads) cannot simultaneously coexist on the same FPGA. To overcome this drawback, some Authors [21] have recently proposed a more general configurable solution that presents the same flexibility of FPGAs but with a context switch time comparable with that CPUs.

After all, in the domain of supercomputing, the current era is referred to as the petascale era, while the next big HPC challenge is to break the exascale barrier. But, due to technological limitations [18] there is growing agreement that reaching this goal will require a substantial shift toward hardware/software codesign [4][15].

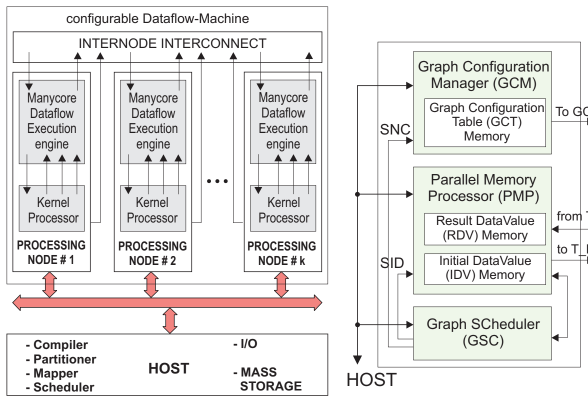
In order to achieve maximum performance, the driving idea is to accelerate the kernel of the application into a computational dataflow engine able to execute its dataflow program graph (DPG) directly in a custom hardware. In this way, the resulting graph execution can not only take advantage from the explosion of the fine-grain spatial parallelism that dataflow presents but can also exploit the temporal parallelism (pipeline) offered by the sequence of actors constituting the graph. Ideally, in the static dataflow form, data can enter and exit each stage of the pipeline at every cycle.

Someone can argue that being a dataflow execution completely asynchronous, this could slow down the run of a pipelined DPG. However this is not a problem for big data computations, since the speed of computation depends on pin throughput and local memory size/bandwidth inside the computational chip [21]. More, it should be considered that, in the many-core scenario, till now none employs single cores as chain of a pipelined computation.

Therefore, these system may seem to perform poorly when compared to systems with a high clock frequeny. However, if the comparison is based on data throughput, these systems perform richly [13] and consume less power, given their ability to perform asynchronous computations and use less space compared to systems driven by a fast clock. Of course, machines based on the dataflow approach perform poorly on relatively simple benchmarks, which are typically not rich in the amount and variety of data structures, but they perform fairly well on relatively sophisticated benchmarks, rich in the amount and variety of data structures.

But, evaluating radically different models of computation

CF'15 May 18-21, 2015, Ischia, Italy
Copyright 2015 ACM 978-1-4503-3358-0/15/05 ...$15.00.
http://dx.doi.org/10.1145/2742854.2747287


configurable Dataflow-Machine
INTERNODE INTERCONNECT
Manycore Dataflow Execution engine
Kernel Processor
PROCESSING NODE # 1
PROCESSING NODE # 2
PROCESSING NODE # k
- Compiler
- Partitioner
- Mapper
- Scheduler
HOST
- I/O
- MASS STORAGE
Graph Configuration Manager (GCM)
Graph Configuration Table (GCT) Memory
To GC
SNC
Parallel Memory Processor (PMP)
Result DataValue (RDV) Memory
from T_O
Initial DataValue (IDV) Memory
to T_I
SID
Graph SCheduler (GSC)
HOST

(1) general architecture (2) Kernel Processor

Figure 1: D^{3}AS system: (1) the computing Dataflow-Machine and (2) the kernel processor

such as dataflow remains yet to be addressed, especially in the context of total cost of ownership [6]. To contribute in this evaluation effort, we show in this work how a computing Dataflow-Machine behaves when it executes the kernel of an application like matrix multiplication and when its execution requires a further acceleration due to the huge amount of input data.

The remainder of the paper is organized as follows. Section 2 summarizes the main characteristics of a Configurable Dataflow-Machine to facilitate the paper reading; Section 3 defines the evaluation model for the case study and shows the resulting performance over the configurable Dataflow-Machine; Section 4 discusses related work in the area of the dataflow architectures; Section 5 for our conclusions.

2. THE CONFIGURABLE DATAFLOW MACHINE AND ITS PROGRAMMING LANGUAGE

2.1 The configurable dataflow machine

A configurable Dataflow-Machine [21], whose general architecture is shown in Fig. 1.1, has been developed within an integrated software-hardware project named Demand Data Driven Architecture System (D^{3}AS) [23]. While D^{3}AS supplies all software activities to compile, partition, map, and create the ordered list of DPGs to assign to each processing, and etc. by means of a host, the configurable Dataflow-Machine (CDM) is constituted by k processing nodes and an internode interconnection for the communication of DPGs allocated over more nodes .

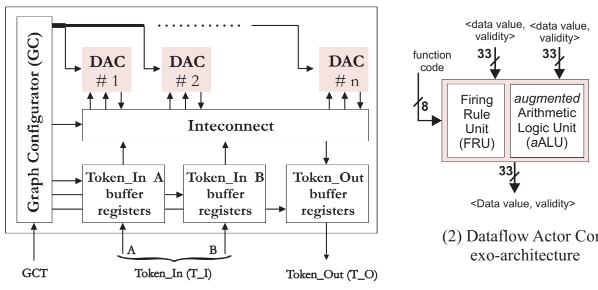
The processing node. It is constituted by a Kernel Processor (Fig. 1.2) that supports the macro-functions to load a DPG onto the associate Many-DAC Dataflow Execution-engine (MDE) (Fig. 2.1). Differently from a von Neumann-based many-core chip, the MDE is a many-core chip where the n DACs (Dataflow Actor Cores) are a set of completely decentralized and self-scheduling execution units connected through a crossbar switch network that implements arcs and links. The graph configurator register holds the DAC function codes and the crossbar-switch code that constitute the DPG configuration loaded onto the MDE. Three I/O register banks are needed to receive/send data tokens from/to the *Kernel Processor*. So, once configured, DACs can fire as soon as valid data tokens start entering the MDE thanks to their asynchronous behavior. Tokens validity determines if a data value is valid or not valid to fire a D# operator instead of using the token presence to perform its functions as it happens in dataflow schemas [3]. A DAC (Fig. 2.2) consists of a firing rule unit that implements the hHLDS firing rules (see Sect. 2.2.1) by means of a simple hardware circuitry, and an *augmented* ALU that implements the D# operator set(see Sect. 2.2.1). In this way, dataflow actor operations are sequenced by the data validity firing rule.
In cooperation with the MDE, the Kernel Processor includes three fundamental blocks whose functionalities are:

- *Graph Configuration Manager* (GCM): (i) all the sub-DPG configuration tables, coming from the compiling phase (on the host), are stored in the Graph Configuration Table memory; (ii) when an enabling Send-Next-Configuration (SNC) signal from the Graph Scheduler (GSC) reaches the graph manager, the graph manager transfers the scheduled graph configuration from its memory to the MDE.

  Parallel Memory Processor (PMP): at the same time, the scheduler sends the enabling Send-Initial-Data (SID) signal to this memory processor that (i) prepares the initial data tokens for their transfer to the dataflow execution engine; after having organized the previous transfer, (ii) prepares the result data tokens for tokens transfer from the dataflow execution engine to the output buffer, as soon as they are ready in the output buffer registers; when the computation ends, it (iii) sends a termination signal to the scheduler.

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

The Internode Interconnect. At the moment, we are evaluating three possible solutions – a crossbar switch, a direct network topology, and a trade-off solution between the first two. The reason is that each of them conditions several parameters like scheduling policy, scalability, cost, performance, and so on [21], when the DPG dimension requires to be sub-DPGs over several processing nodes.



(1) Manycore Dataflow Execution Engine

Figure 2: D^{3}AS computing system: the execution engine

2.2 *h*HLDS and the programming language

2.2.1 The *h*HLDS model

High-Level Dataflow System (HLDS) [22] is a formal model to describe the behavior of a directed dataflow graph where nodes are operators (actors) or links (places to hold tokens) that can have heterogeneous I/O conditions. Nodes are connected by arcs along which tokens (data and control) may travel. In this paper, also the *homogeneous* HLDS (*h*HLDS) model was presented. *h*HLDS describes the behavior of a static dataflow graph imposing homogeneous I/O conditions on actors but not on links. Actors can only have exactly one output and two input arcs and consume and produce only data tokens; links represent only connections between arcs. Since *h*HLDS' actors cannot produce control tokens, merge, switch, and logic-gate actors [3] are not present. In this model, while actors are determinate, links may be not determinate. In the *h*HLDS model there exist two types of links: i) Joint links, which represent a place where two or more output arcs can coexist and ii) Replica links, which are similar to joint links but have only one output arcs. In the case of Joint links the output arc (among the several available) – where the token will travel – is unpredictable. In contrast, these features simplify the design of a dataflow execution engine chip using only identical DACs and one type of connection among them. In addition, despite the model simplicity, in *h*HLDS it has been proved that it is always possible to obtain DPGs which are determinate and well-behaved, and where:

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

In *h*HLDS actors and links are connected to form a more complex DFG. However, the resulting DFG may be non-determinate if cycles occur because no closure property can be guaranteed [14]. This happens for sure when the graph includes joint links, which are not-determinate. In the case the DPG is determinate and well-behaved, it is named *macro-Actor* (mA), and it is characterized by having I(mA)>2 and O(mA)≥ 1 where I(mA) is the number of input arcs (in-set) of the mA and O(mA) is the number of its output arcs (out-set).

2.2.2 The D# machine language

At the present, the machine offers programming in both Chiara functional language [20] and the machine language called D#. D# is both the machine language and the graphical representation language that describes the dataflow graph of a program. In the D# language a program is a collection of standard expressions that form a DPG where each expression refers an actor and specifies its functionality. A D# particularity is that its operators form the functionally complete set of the elemental operators for Chiara, allowing thus a DPG be also represented graphically. It has been defined applying the demand-data-driven approach to co-design methodology [24] between the functional paradigm and the *h*HLDS paradigm.

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

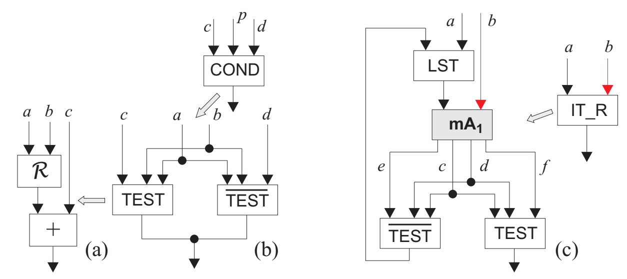


Figure 3: The basic macro-Actors (mAs) in D#: (a) TEST, (b) COND, (c) IT_R

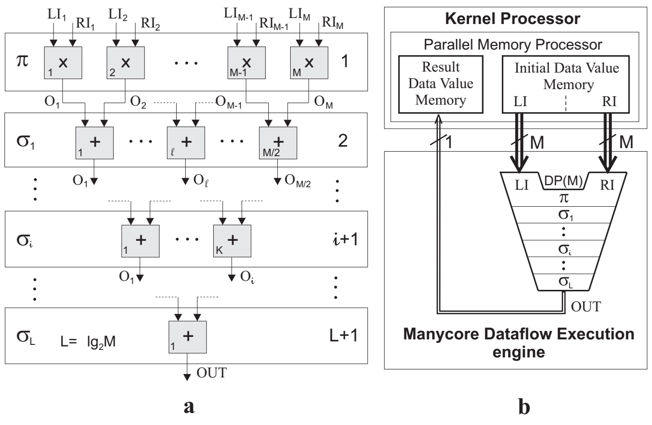
As macro-Actors (mAs) structures in D# are formed like in *h*HLDS, here we shortly report the fundamental characteristics of those that allow the creation of more complex structures (i.e. TEST, COND, and IT_R mAs).

The macro-Actor TEST. It is the simplest relational structure of data-dependent DPG. TEST is a mA with in-set = 3 and out-set =1 and is formed connecting the relational actor $\mathcal{R}$ to the actor that performs the arithmetic operator $+$ as shown in Fig 3(a). If a and b satisfy the relation $\mathcal{R}$, the token generated is a valid token with data-value 0 (zero), and the token c is produced. If its relation is not satisfied, the token generated is not valid with data-value "don't-care". When coupled to its complement $\overline{\text{TEST}}$, it forms a fundamental building-block to create conditional and iterative mAs.

The macro-Actor COND. It is the simplest conditional structure. COND is is a mA with in-set = 4 and out-set =1 and is formed connecting the two mAs TEST and $\overline{\text{TEST}}$ with a Joint link as shown in Fig. 3(b). If the relation of TEST is satisfied, it produces the token c, otherwise it produces the token d. It forms the building-block to create more complex conditional structures as an example the construct *case*.

The macro-Actor IT_R. It is the iterative data-depend structure. IT_R is a mA with in-set = 2 and out-set =1 and is formed connecting the two mAs TEST and $\overline{\text{TEST}}$, an arithmetic actor or a macro-Actor mA_1, and the actor

[1]Validity is the intrinsic characterization of a token, and operations start or o not on this information. In contrast to Dennis' *token presence* [3], where operations are triggered by the presence of tokens, with the validity information we not only remove the check of token absence on any output arc to fire an actor but remove also the *Instruction Scheduler*.



π
σ1
σi
σL
L= lg2M
OUT
a
Kernel Processor
Parallel Memory Processor
Result Data Value Memory
Initial Data Value Memory
LI
RI
M
DP(M)
Manycore Dataflow Execution engine
b

Figure 4: Matrix product: (a) the generalized dataflow program graph and (b) the corresponding resources required to a processing node

LST as shown in Fig. 3(c). The LST semantics is: it selects the right token the first time it is fired, the left token otherwise. It constitutes the building- block to create more complex data-dependent iterative structures. As an example, if mA1 (at the center of Fig. 3(c)) is itself an IT_R macro-Actor, the figure represents a determinate and well-behaved nested-data-dependent iterative structure.

3. CASE STUDY: THE MATRIX MULTIPLICATION

To test some computational capabilities of the configurable Dataflow-Machine we have used the product of two matrices $C(N,P) = A(N,M) \times B(M,P)$ where $a_{i,j}$ with $i = 1,2,\dots,N$ and $j = 1,2,\dots,M$ is an element of A, and $b_{j,k}$ with $j = 1,2,\dots,M$ and $k = 1,2,\dots,P$ is an element of B. **The evaluation model**. Fig. 4.a shows the generalized DPG for the matrix product expressed in terms of dot product $(\underline{\mathbf{a}}_i \cdot \underline{\mathbf{b}}_i)$ of the N row vectors of matrix A

$$\underline{\mathbf{a}}_i = \{a_{i1}, a_{i2}, \dots, a_{iM}\} \qquad i = 1,2,\dots,N$$

and P column vector of matrix B

$$\underline{\mathbf{b}}_i = \{b_{1i}, b_{2i}, \dots, b_{Mi}\} \qquad i = 1,2,\dots,P$$

As we can observe, it constitutes a reverse tree organized in $L_s = \lceil log_2 M \rceil + 1$ sequential levels. It consists of $2M - 1M$ actors with M of them implementing the multiplication function and $M-1$ of them implementing the addition function. LI, RI, and O represent the left input token set, the right input token set, and the output token set respectively, as they are defined in the language D#.
Fig. 4.b shows how a generalized processing node[2] is organized in terms of the MDE engine and kernel processor. Inside the MDE engine, all these DACs are allocated on a number of levels like in the DPG and execute the dot product computation in a number of sequential stages $L_n = L + 1 = \lceil log_2 M \rceil + 1$.
At each stage, the maximum spatial parallelism is exploited,

[2]In this context, without losing of generality, we use the term node processing instead of generalized processing node to indicate the set of processing nodes necessary to represent the whole DPG.

Table 1: DPE engine characteristic table

Level	DAC number	Operation type	time	Parallelism degree (spatial)
1	M	mpy	τ_m	M
2	M/2	add	τ_a	M/2
...	...	...	...	...
L_n	1	add	τ_a	1
Total computing time $T_c = \tau_m + \lceil log_2 M \rceil * \tau_a$				

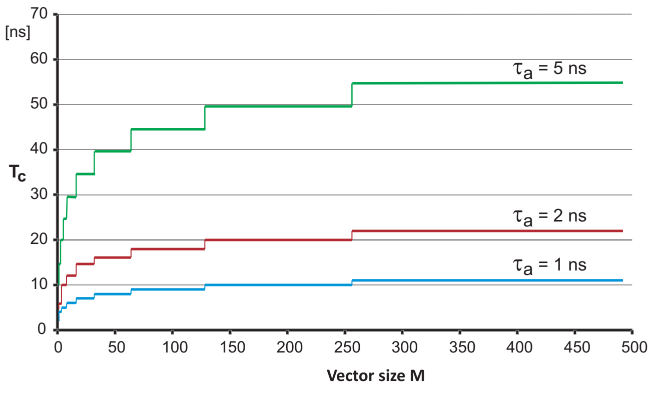


Figure 5: T_c versus vector size M for different values of τ_a

and the total computational time is $T_c = \tau_m + \lceil log_2 M \rceil * \tau_a$ where τ_m is the multiplication delay and τ_a is the addition delay. As these delays depend on the utilized technology, we assume that $\tau_m = 2\tau_a$ and that $1ns \leq \tau_a \leq 5ns$, including the token transfer from the kernel processor.
Its characteristics are synthetically shown in Table 1. Figure 5 shows the total computation time T_c of the DPE versus vector dimension values up to 500 for three different values of τ_a.
It is important to outline that, with reference to the DPG, it is possible to consider this operation decomposed into a linearly organized set of tasks or phases. Besides, in a MDE, the pipelining technique may be used to overlap the computation of independent sets of operands because it consists of a number of stages equal to the number of levels of the calculation tree $L_n = L + 1$. A first stage π executes the parallel multiplication of the components of the two M-dimensioned vectors; L stages execute the parallel additions σ_i with $i = 1,2,\dots,L$ to compute the sum of the M products. The interstage latches needed for the correct operations of pipeline are inside the Initial Data Value memory of kernel processor. This is a purely asynchronous pipelined system,where the times at which significant events take place, including the flow of tokens between stages, are not under the control of a central clock but are regulated in a completely decentralized manner by the DACs belonging to the different stages.
For each stage i, we define a "stage delay" τ_i as the time required to perform it. The "pipeline period" τ_p, the minimum time interval between two successive stages of the pipeline, is determined as $\tau_p = Max(\tau_i)$ for $1 \leq i \leq L_n$. The "pipeline frequency" $\phi = 1/\tau_p$, the inverse of the pipeline period, is the frequency at which the pipeline operates. The characteristics of pipelined MDE are summarized in Table 2.

Table 2: Pipelined MDE engine characteristics.

Number of stages L_n	:	$\lceil log_2 M \rceil + 1$
Interstage latches	:	inside the DAC
Stage delay τ_i	:	time required for the i^{th} stage to carry out its task
Pipelining period τ_p	:	the minimum time interval between two successive stages of the pipeline $\tau_p = Max(\tau_i) 1 \leq i \leq n$
Pipelining frequency ϕ	:	$1/\tau_p$
Type of parallelism	:	spatial and temporal
Parallelism degree	:	$2M - 1$

To evaluate the performance enhancement due to the pipelining, a number of performance characteristics should be considered as the throughput, the speedup, the throughput rate, and latency and efficiency. Among these the throughput TP and the speedup S are the most important and meaningful for our purpose. For the MDE operating in non-pipelined mode, the number of stages completely processed by the system per unit time is

$$TP = \frac{1}{\tau_m + \lceil \log_2 M \rceil * \tau_a}$$

In the pipelined mode (Table 2) the MDE operates as an L_n-stage pipeline system where the tasks are delivered to the pipeline at the same time interval τ_p. The system will require a time $L_n \times \tau_p$ to fill the pipeline and process the first task; thereafter, it will process one task after every time interval τ_p. Therefore, the pipeline system processes K tasks (stages) in a time

$$T_{cp}(K, L_n) = (\lceil log_2 M \rceil + K) * \tau_p$$

Hence the throughput TP_p of the system in pipeline mode is

$$TP_p = \frac{K}{(\lceil log_2 M \rceil + K) * \tau_p}$$

The speedup S of the pipelined L_n-stage computing K tasks respect to the non-pipelined computation is

$$S = \frac{K * (\tau_m + \lceil log_2 M \rceil) * \tau_p}{(\lceil log_2 M \rceil + K) * \tau_p}$$

Performance evaluation. To evaluate the behavior of the configurable Dataflow-Machine, we have used CODACS demonstrator [19], an FPGA-based prototype based on 5 Altera APEX20K15-3C components. The main performance characteristics of the MDE for the pipelined and non-pipelined mode operating modalities are reported in Table 3. Figure 6 shows Performance, in Gigaflops, of the MDE versus vector size M for $\tau_a = 1$ ns.
At this point, parameterized to the problem size M, the MDE has been completely characterized and defined from both an architectural and performance point of view. This is very important because it constitutes the *hard* part of the dataflow process for the dot products of two vectors of size M. In this case all the K entries of the product matrix C are the dot product of row and column vectors of the two matrices A and B, so the dot products to execute are of this type.

Table 3: Characteristics of a pipelined vs non-pipelined MDE.

	Operating Modalities	
	Non Pipelined	Pipelined
Throughput	$\frac{1}{\tau_m + \lceil \log_2 M \rceil * \tau_a}$	$\frac{K}{(\lceil \log_2 M \rceil + K) * \tau_p}$
Speedup	——	$\frac{K * (\tau_m + \lceil \log_2 M \rceil \tau_a)}{(\lceil \log_2 M \rceil + K) * \tau_p}$
Performance	$\frac{2M - 1}{\tau_m + \lceil \log_2 M \rceil \tau_a}$	$\frac{2M - 1}{\tau_p}$

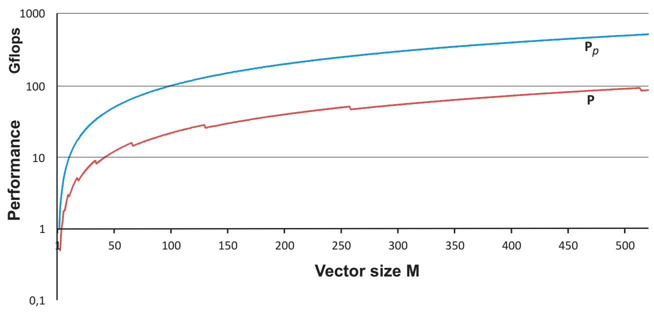


Figure 6: MDE performance in pipelined (P$_p$) and non pipelined (P) modes versus vector size M ($\tau_a = 1$ ns)

4. RELATED WORK

A recent project that investigated how to exploit dataflow concepts in many-cores was TERAFLUX [8], that introduced dynamic dataflow based threads called DF-threads [9]. Its challenging goal was to develop a dataflow based execution model on standard off-the-shelf cores [10][11]. Additional analysis lead to exploration of execution over faulty components [25] and applicability to the Haskell and Transactional Memory [7]. A step toward the fine granularity of the dataflow is the *dataflow codelet* or simply *codelet* [26]. A codelet is a collection of machine instructions smaller than a thread but coarser than the traditional dataflow and represents the finest granularity of parallelism that can be scheduled as a unit of computation. Its operational semantics asserts that a codelet is first enabled, when all its events are satisfied, and then fired (scheduled), when a processing element becomes available. Less recently, new reconfigurable architectures very similar to the dataflow approaches have been proposed. TRIPS [1] is based on a hybrid von Neumann/dataflow architecture that combines an instance of coarse-grained, polymorphous grid processor core with an adaptive on-chip memory system. TRIPS uses three different execution modes, focusing on instruction-, data- or thread-level parallelism. WaveScalar [17], on the other hand, totally abandons the program counter. Both TRIPS and WaveScalar take a hybrid static/dynamic approach to scheduling instruction execution by carefully placing instructions in an array of processing elements and then allowing execution to proceed dynamically. But, in the configurable Dataflow-Machine during the execution of an algorithm it is not necessary to fetch any instruction or data from memory thanks to its dataflow soft-core. Another lively interest in dataflow comes from FPGAs-based computing. In this field, there is a lot of research on the mapping and execution of dataflow graphs [2] [16], but no solution is addressed to their execution in hardware without employing the control flow information. Besides, the reconfiguration time cost when a DPG changes is beyond compared to our MDE.

5. CONCLUSIONS

This work outlines how a manycore dataflow execution engine design in terms of architecture, machine language, and dataflow model, is feasible to exploit all the fine grain parallelism present in the algorithms whose execution requires a huge amount of input data. In this contest, other than the utilization of Dataflow Actor Cores, a fundamental role is played by the co-design approach between the static dataflow model of execution hHLDS and the machine language D#. Since the machine behavior is totally asynchronous, its execution shows interesting performance because the computation results purged by unnecessary instructions present on power-hungry von Neumann-style softcores. Power savings are expected by the adoption of this asynchronous approach.

Acknowledgements

We thank the anonymous reviewers for their useful comments. This work is partly supported by the European Projects TERAFLUX (id. 249013) and AXIOM (id. 645496).

6. REFERENCES

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

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

[3] J. Dennis, J. Fosseen, and J. Linderman. Data flow schemas. In A. Ershov and V. A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*, pages 187–216. Springer Berlin, 1974.

[4] S. Dosanjh, R. Barrett, M. Heroux, and A. Rodrigues. Achieving exascale computing through hardware/software co-design. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 6960 of *Lecture Notes in Computer Science*, pages 5–7. Springer Berlin, 2011.

[5] G. Estrin. Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer. *Annals of the History of Computing, IEEE*, 24(4):3–9, Oct 2002.

[6] M. J. Flynn, O. Mencer, V. Milutinovic, G. Rakocevic, P. Stenstrom, R. Trobec, and M. Valero. Moving from petaflops to petadata. *Commun. ACM*, 56(5):39–42, May 2013.

[7] R. Giorgi. Accelerating haskell on a dataflow architecture: a case study including transactional memory. In *Proc. Int.l Conf. on Computer Engineering and Application (CEA)*, pages 91–100, Dubai, UAE, feb 2015.

[8] R. Giorgi and et al. Teraflux: Harnessing dataflow in next generation teradevices. *ELSEVIER Microprocessors and Microsystems*, Apr 2014.

[9] R. Giorgi and P. Faraboschi. An introduction to df-threads and their execution model. In *IEEE Proceedings of MPP-2014*, pages 60–65, Paris, France, oct 2014.

[10] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *Future Generation Computer Systems*, (0):–, 2015.

[11] N. Ho, A. Mondelli, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Enhancing an x86_64 multi-core architecture with data-flow execution support. In *ACM Proc. of Computing Froniers*, pages 1–2, Ischia, Italy, May 2015.

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

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

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

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

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

[17] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007.

[18] M. Y. Vardi. Is moore's party over? *Commun. ACM*, 54(11):5–5, Nov. 2011.

[19] L. Verdoscia. Codacs project: A development tool for embedded system prototyping. In Z. Wu, C. Chen, M. Guo, and J. Bu, editors, *Embedded Software and Systems*, volume 3605 of *Lecture Notes in Computer Science*, pages 59–64. Springer Berlin, 2005.

[20] L. Verdoscia, M. Danelutto, and R. Esposito. CODACS prototype: Chiara language and its compiler. In *Proceedings of the First International Workshop on Embedded Computing*, Tokyo University of Technology, Hachioji, Tokyo, Japan, Mar. 23–26, 2004. IEEE Computer Society Press.

[21] L. Verdoscia, R. Giorgi, and R. Vaccaro. A clockless computing system based on the static dataflow paradigm. In *International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2014), in conjunction with PACT 2014*, Edmonton, Alberta, CA, Aug.24 2014.

[22] L. Verdoscia and R. Vaccaro. A high-level dataflow system. *Computing*, 60(4):285–305, 1998.

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

[24] L. Verdoscia and R. Vaccaro. Position paper: Validity of the static dataflow approach for exascale computing challenges. In *International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2013), in conjunction with PACT 2013*, Edinburgh, Scotland, UK, Sept. 8, 2013.

[25] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. Architectural support for fault tolerance in a teradevice dataflow system. *Springer International Journal of Parallel Programming*, (0):1–25, May 2014.

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