NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

The Dataflow Computational Model And Its Evolution

Panagiotis G. Repouskos

Supervisor: **Panagiotis Rondogiannis,** Professor NKUA

ATHENS

MAY 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Το υπολογιστικό μοντέλο Dataflow και η εξέλιξη του

Παναγιώτης Γ. Ρεπούσκος

Επιβλέπων: **Παναγιώτης Ροντογιάννης,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΪΟΣ 2017

BSc THESIS

The Dataflow Computational Model And Its Evolution

Panagiotis G. Repouskos
S.N.: 1115201000106

SUPERVISOR: **Panagiotis Rondogiannis,** Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Το υπολογιστικό μοντέλο Dataflow και η εξέλιξη του

Παναγιώτης Γ. Ρεπούσκος
Α.Μ.: 1115201000106

ΕΠΙΒΛΕΠΩΝ: Παναγιώτης Ροντογιάννης, Καθηγητής ΕΚΠΑ

ABSTRACT

The dataflow computational model is an alternative to the von-Neumann model. Its most significant aspects are, that it is based on asynchronous instructions scheduling and exposes massive parallelism. This thesis is a review of the dataflow computational model, as well as of some hybrid models, which lie between the pure dataflow and the von Neumann model. Additionally, there are some references to dataflow principles, that are or are being adopted by conventional machines, programming languages and distributed computing systems.

SUBJECT AREA: Dataflow Model

KEYWORDS: dataflow computational model, dataflow programming, dataflow architecture, control-flow, semantics, data driven, demand driven, asynchronous scheduling, von Neumann, distributed computing, programming languages, coordination languages, parallel programming, implicit parallelism

ΠΕΡΙΛΗΨΗ

Το υπολογιστικό μοντέλο dataflow είναι ένα εναλλακτικό του von-Neumann. Τα κυριότερα χαρακτηριστικά του είναι ο ασύγχρονος προγραμματισμός εργασιών και το ότι επιτρέπει μαζική παραλληλία. Αυτή η πτυχιακή είναι μία μελέτη αυτού του μοντέλου, καθώς και μερικών υβριδικών μοντέλων, που βρίσκονται ανάμεσα στο αρχικό μοντέλο dataflow και στο von-Neumann. Τέλος, υπάρχουν αναφορές σε μερικές αρχές του dataflow, οι οποίες έχουν υιοθετηθεί σε συμβατικές μηχανές, γλώσσες προγραμματισμού και συστήματα κατανεμημένων υπολογισμών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μοντέλο Dataflow

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: υπολογιστικό μοντέλο dataflow, προγραμματισμός σε dataflow, αρχιτεκτονικές dataflow, control-flow, σημασιολογία, data driven, demand driven, ασύγχρονος προγραμματισμός, von Neumann, κατανεμημένα συστήματα, γλώσσες προγραμματισμού, γλώσσες συντονισμού, παράλληλος προγραμματιμός, παραλληλία

ACKNOWLEDGEMENTS

I would like to thank Professor Panagiotis Rondogiannis, who has become a source of inspiration for theoretical computer science in the Department of Informatics and Telecommunications. It was through his courses ”Principles of Programming Languages” and ”Semantics of Programming Languages”, that I first got acquainted with the declarative programming paradigm and aware of the underlying mathematics behind programming languages.

CONTENTS

LIST OF FIGURES

PREFACE

In recent years, research on dataflow has been diminished Lately however, interest in the dataflow principles and in how they can be used in other areas, has spiked, especially in the field of big data.

The aim of this thesis is, to serve as a handbook of the most important research on the dataflow model and its variants. In particular, it provides an introduction of dataflow to individuals unfamiliar with this model - something new, since all the surveys on dataflow presuppose a certain familiarity. Lastly, some examples are given, of how dataflow principles have influenced and how they could influence other fields.

1. INTRODUCTION

The dataflow computational model is an alternative to the von-Neumann model. In the von Neumann model a program is represented as a sequence of instructions, and at each step of the execution, the program counter determines which instruction will be executed next. In dataflow, a program is represented as a directed graph, and there does not exist a complete order in the execution of the instructions. Many instructions could be executed simultaneously. As such, there are fundamentally different evaluation methods.

The aim of this survey is to provide insight into all these novelties of dataflow, without presupposing any knowledge on behalf of the reader. This is the structure of the thesis:

In the second chapter, the reader is introduced to the notion of computational models and to how each of these models defines a unique program representation and computation organization.

In the third chapter, dataflow graphs are presented. There are several ways to define a dataflow program, and each definition comes with its own architecture and evaluation method. However, all these evaluation methods have one principle in common; they are data driven, i.e. all instructions that can be executed, are executed.

In the fourth chapter, operator nets are presented. Operator nets are a special type of data-flow graphs, with their own semantics. They are evaluated in a demand driven manner, that is to say, no redundant computations are made. This allows for more expressiveness than in data driven dataflow.

In the fifth chapter, a comparison is made between dataflow and von Neumann style. Each model has its own strengths and weaknesses; in dataflow massive parallelism is exposed, while in von Neumann sequential code is executed very efficiently. There have been many attempts to combine these two models, and some of these hybrids are presented.

In the sixth chapter, some dataflow languages are presented, as well as some influences of dataflow in conventional procedural languages.

In the seventh and last chapter, we see how dataflow can provide a formal framework for distributed computing systems. Some of the newest ones, that have adopted many of the dataflow principles, are briefly mentioned.

2. COMPUTATIONAL MODELS

In this thesis, we describe a different approach to computing and computer architecture; the *dataflow* approach. Before we start examining this model, in which parallelism is naturally and implicitly expressed, it would be useful to juxtapose it and the von Neumann style and focus on how different ways to evaluate programs can affect our approach to architecture and program organization [73].

2.1 Synopsis of Computational Models

2.1.1 Control Flow

The *control flow* is a model familiar to all, especially the traditional von Neumann control flow, with a single thread of control.

A program is represented as a series of instructions, each one consisting of an operator and operands, which are either literals or references. After an instruction has been executed, the result is stored in memory and control is passed implicitly in the next instruction in sequence. There are of course ways to explicit transfer control to a specific instruction, e.g. with the *GOTO* command.

There are also parallel flow models, with more than one thread of control, as for example in *Unix* through the *fork* system call. The system must also provide ways to synchronize these control threads, e.g. *semaphores*. But even in parallel control flow, the flow of control is implicitly sequential.

2.1.2 Data Flow

In *data flow*, the programs are represented as directed graphs. Vertices denote instructions and arcs represent data dependencies. An instruction is executed when it becomes *enabled*, that is when all arguments are known, or more concretely when it has sufficient data on all its input arcs. Several instructions can be executed at any time.

There are three key differences from the control flow model:

1. Flows of control are tied to the flow of data; *data availability* determines which instruction(s) will be executed next.
2. *Partial results* are directly passed from one instruction to its successors as data tokens.
3. There is no concept of *shared memory*. When a token is consumed by a node, it is removed from the input arc, and its value is no longer available as input for any other instruction.

2.1.3 Reduction

In reduction, programs are built from nested expressions and are mathematically equivalent to their result. For example, consider the following *definition*:

$$a = (5 \cdot b) + 9$$

A *demand* for the result of "*a*" is a demand for "*a*" to be written in a simpler form. Conceptually, we can view this evaluation as function application, i.e. calling the definition of "*a*". To avoid unwanted indeterminacy, we allow only one definition for "*a*" (*single assignment* rule), so every demand for it yields the same result (*referential transparency*). Each instruction, when being executed, can manipulate a separate copy of the definition (*string reduction*), or all instructions accessing a particular definition, can manipulate references to that definition (*graph reduction*).

As an example, lets examine the string reduction program for the previous definition. To make it more apparent that the definition is built from nested expressions, let's rewrite it at an instruction level, in prefix notation:

$$\begin{aligned} a &= (+\ x\ 9) \\ x &= (\cdot\ 5\ b) \end{aligned}$$

and assume that *b* holds the value 4. When a demand for the value of "*a*" arrives, we overwrite the reference of "*a*" with its definition:

$$a \Rightarrow (+\ x\ 9)$$

Now, we need to create a demand for the value of *x*:

$$(+\ x\ 9) \Rightarrow (+\ (\cdot\ 5\ b)\ 9)$$

And one more step for the value of *b*:

$$(+\ (\cdot\ 5\ b)\ 9) \Rightarrow (+\ (\cdot\ 5\ 4)\ 9) \Rightarrow 29$$

Essentially, when we want to compute a function, we create demands for its missing arguments. The key points to note are:

1. A program is built from nested expressions (instructions, arguments, etc. are considered expressions).
2. There is no notion of an *update* operation.
3. The sequencing constraints are those implied by demands.
4. A demand does not necessarily return a value; it may as well return a complex argument, e.g. a function as input to a higher order function.

2.2 Computation Organization

After describing some basic principles of these three major computing models, we should examine how the computation itself progresses, i.e. how the sequencing and execution of instructions affect and define the computation, organizing it in a series of successive states.

The phases we are interested in are the *selection* phase, where the set of instructions for possible execution is determined, the *examination* phase, in which it is determined if an instruction can be executed (firing rule in the terminology of dataflow, as in Chapter 3), and the actual execution phase.

2.2.1 Control Driven

This computation organization belongs in the control flow model. At every stage, the program counter selects, which instruction will be executed next and it happens automatically. The results are communicated through *shared memory* and the stages of the computation are represented by the contents of this shared memory.

2.2.2 Data Driven

Data driven or *availability driven* denotes computation organizations, such as those described in Chapter 3.

Conceptually, we can imagine that every instruction (node) has a dedicated PE. The examination phase consists of a firing rule; we will examine some variations, but all of them have the following principle in common:

An instruction can be executed, if all its arguments are available. Otherwise, it remains dormant.

The result is not stored in memory, but is passed directly to the instruction's successors.

The advantage of data driven computation is the parallelism it exposes. Two disadvantages are the overhead due to the asynchronous parallelism [26] and the computation of perhaps unneeded arguments.

2.2.3 Demand Driven

Demand driven denotes the computation organization, in which an expression is evaluated, only if a demand has been created for the value it produces. If the instruction can be executed, it is. If some argument is missing, then the system creates a demand for it.

The advantage in demand driven computation is that only what is needed to be computed to produce the final result, is computed [43]. Unneeded instructions are not executed. A disadvantage is that the propagation of demands can be expensive.

3. DATA DRIVEN DATAFLOW

3.1 Abstract Dataflow

3.1.1 Acyclic Dataflow Graphs

In the pure dataflow execution model, a program is represented by a directed graph [4, 31], where the nodes (also known as *actors*) denote primitive instructions, e.g. arithmetic or comparison operations, and the edges (or *arcs*) denote *data dependencies* between operations. As an example, Figure 3.1 shows the acyclic dataflow program graph for the following expression. The program is written following the syntax of the dataflow language Lucid [75].

```
(x + y) / c
where
     x = a − b;
     y = 4 ∗ c;
end
```

Conceptually, arcs behave as unbounded first-in, first-out (FIFO) queues, along which data travels as *tokens*. *Input arcs* are those flowing towards a node, while those flowing away, are called *output arcs*. A node executes (or *fires*) when a token is available in each input arc (*firing set*). Upon firing, a token is consumed from each input arc, a result is computed and a token containing this result is produced on each output arc. After firing, the actor waits till tokens are once again available in all input arcs. When an arc forks, the token is copied and sent to both directions. A program terminates, when there is no longer an actor that can fire.

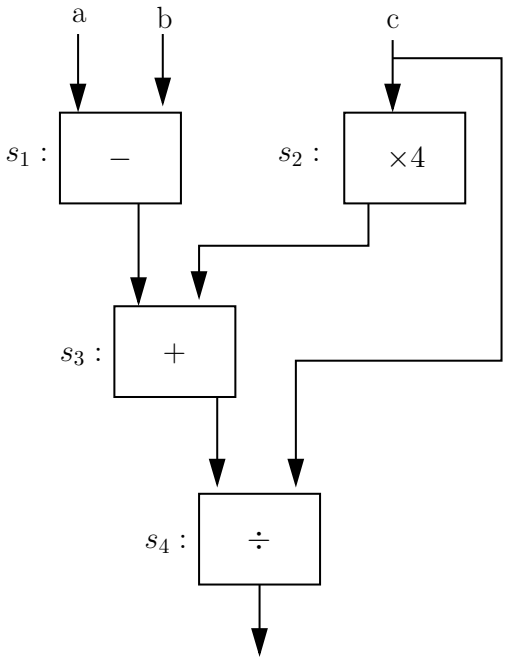

Figure 3.1: An Acyclic Dataflow Graph

Notice that by scheduling instructions for execution based on the availability of operands (*data-driven evaluation*), unless implied by data dependencies, there is no strict order to their execution; one may fire before an other or they may fire simultaneously. For instance, in Figure 3.1 the nodes s_1 and s_2 have no data dependency between them, so they can fire simultaneously. However, since s_3 is data dependent on s_1 and s_2, it must wait for their execution to end. This stands in contrast to the classic von Neumann model, where the scheduling of instructions for execution is decreed by the program counter, leading to prescribing a specific execution order, inherent to assignment based programs.

Thus, the first key property of the dataflow approach comes to light: *massive parallelism*. The implied parallelism by using data dependencies as a scheduling mechanism, can clearly provide a substantial speed improvement; two nodes, unless there is an explicit data dependency between them, may fire simultaneously (*spatial parallelism* or *structural parallelism*). Furthermore, if we provided many sets of input (e.g. many values for a, b, c in Figure 3.1), computations for the second evaluation of the program could begin before those for the first evaluation finished. This is known as *temporal parallelism* or *pipelined dataflow*.

Another key point is *determinacy*; the results are independent from the order in which potentially parallel nodes fire, or more concretely, for a given set of inputs, a program will always produce the same set of outputs [4, 31, 50]. What leads to this property is that data tokens travel in ordered queues and that *the operation of every actor is functional* - a result of an operation is purely a function of its input values. This is because data is never modified (new data tokens are created as output of a firing), so the nodes are side-effect free and because of the absence of a global data store there is locality of effect. In concurrent execution, determinacy leads to further speed improvements, since concurrency-reducing synchronization to avoid time dependent errors is not required. However, determinacy, although a desired property, is in itself a restriction [31]. For instance, a rooms booking service, can give the last room to only one customer, even if many apply for it at the same time. Some research done on nondeterminate behavior will be presented later.

3.1.2 Conditional and Loop Dataflow Graphs

In order to build conditional and loop program graphs, we need to introduce two control operators: *switch* and *merge*. Their usage is demonstrated through the following examples, found in [4]. First, consider the following expression and its equivalent program graph in Figure 3.2:

$$\textbf{if } x < y \textbf{ then } x + y \textbf{ else } x - y$$

The initial input tokens provide data input to the switches and to the predicate operator. In turn, the predicate operator yields a boolean value which serves as control input to all switches and merges. The switch operator routes its data input to the appropriate arc, according to its control input's value. The merge operator, has two data input arcs labeled True and False and a control input. According to the control input's value, it reproduces the data input token from the True or False side to its output arc.

To realize iterations, we must introduce cycles in the program graphs. The program in Figure 3.3 computes the sum $\sum_{i=1}^{N} F(i)$. The dotted lines represent the output of the predicate operator and the "blob" with F is a black box that computes the function. The initial values

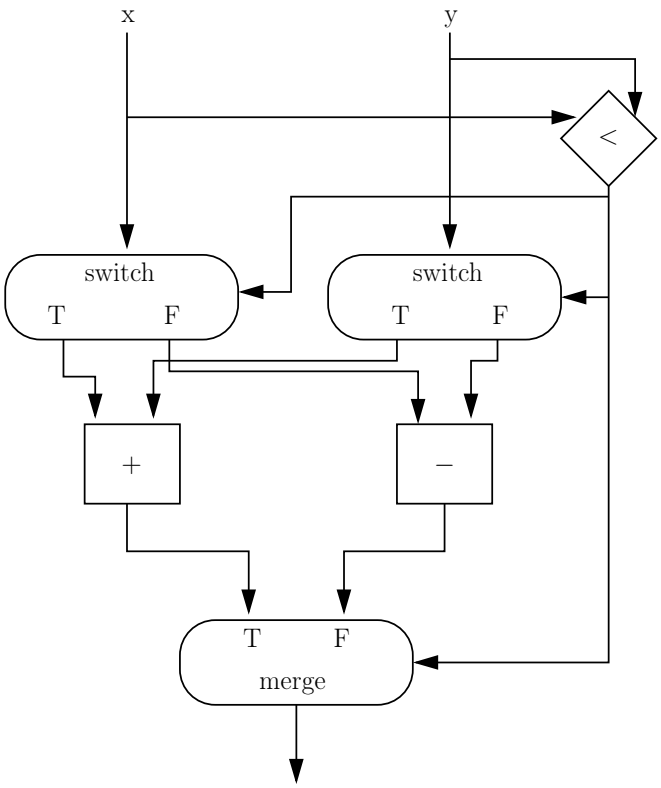

Figure 3.2: A Conditional Graph

of i and sum are given as input to the program. While the predicate evaluates to True, the data tokens are routed back to the loop body.

Note that if the function F requires a lot of time for computation, the tokens with the index variable i will continue circulating, possibly causing more computations of F to be initiated. In other words, unrelated iterations of the same loop body can be executed simultaneously. This behavior is also known as *dynamic unfolding of a loop*.

3.1.3 Data Structures

So far, we have only examined tokens that denote simple values and not complex data structures. Although *fully general in computational expressivity* [45], this lack in storage expressivity limits its practical use.

In principle, we may think of tokens as denoting arbitrarily large data structures and define nodes that consume and produce such tokens. However, we can think of data structures in this way only in the abstract model. Practically this approach is infeasible. As we examine more approaches to dataflow, we will introduce new ways to handle data structures.

We should emphasize that data structures in dataflow languages must not be treated as those in conventional assignment languages, such as C or Java. In a dataflow program, structures are initialized, read and modified in a distributed fashion. Some points we should take into account for our approach to data structures:

- Suppose that a structure modification capability was present in the dataflow language and two tokens carried pointers to the same structure and initiated a modification and a read operation respectively. A time synchronization problem will emerge,

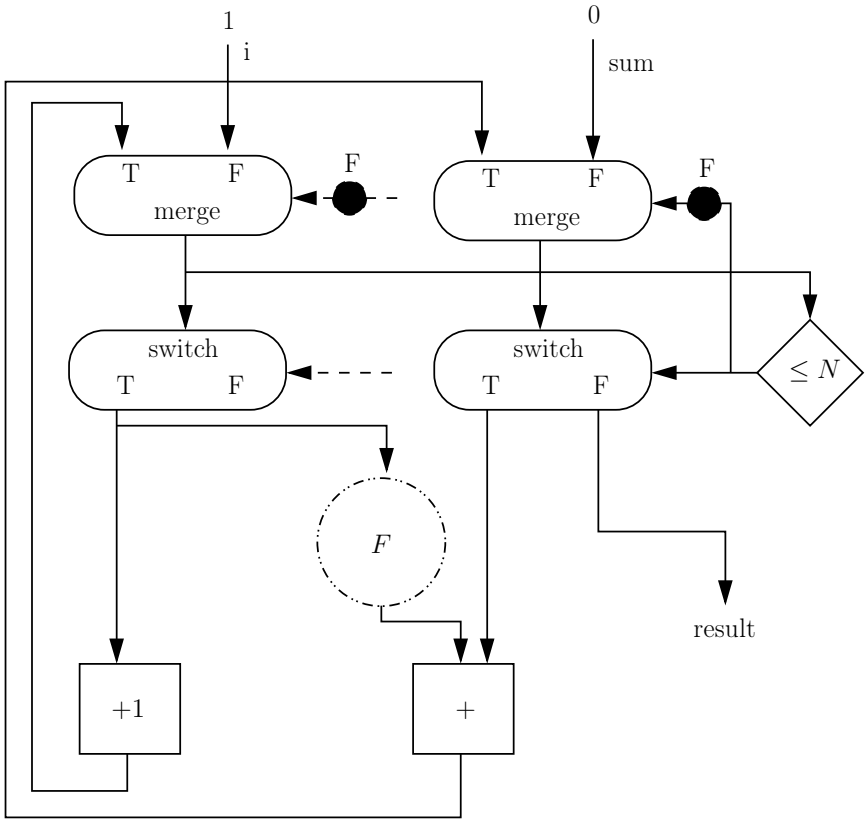

Figure 3.3: A Loop Graph

as the order of their execution is merely prescribed by data dependencies. This would eliminate the determinacy of the model.

- The most attractive characteristic of dataflow is parallelism. The capability to perform operations in parallel to a structure is important. For example, consider the following program:

```
for i=1 To n Do {
    A[i] = f(i);
}
```

  Assuming that *f* denotes a functional operation, all the updates can occur concurrently. A one-operation-at-a-time restriction on the array, eliminates the parallelism that can be exploited. Furthermore, we would like to be able to read the value of the first cell it has been produced, even if the thousandth cell has yet to be computed.

  Actually, building on the previous point, since different parts of the program may be active at a time, a request for a read operation at the first cell may arrive, before the computation of the first cell itself has even begun.

- In the dataflow computational model, the meaning of *update* is not present. This would imply, that when we want to update a part of an array, we have to recreate the whole array. Such a memory-hungry approach has obvious drawbacks.

3.1.4 Discussion

The dataflow computing model examined so far, was done so solely from a theoretical viewpoint. When we try to implement it, we find out that no implementation can exactly mirror it. First, it assumes unbounded FIFO queues on the arcs. Perhaps, it is not as intuitive as in the von Neumann model, where the notion of assignment exists, but *tokens imply storage*. So, to support this model we require unbounded memory, which is impossible. Second, the pure dataflow model assumes that any number of actors can fire in parallel, while clearly in any hardware implementation the PEs (processing elements) will be finite. Concretely, the system may deadlock while the dataflow model predicts no deadlock. Last, difficulties were also presented in trying to implement the FIFO behavior of the nodes.

3.2 Static Dataflow

To remedy the aforementioned problems, two different approaches were researched. The *static dataflow* architecture was introduced by Dennis and Misunas [32].

As described in Subchapter 3.1.1, the decision, of when an actor is ready to fire, is based on the following simple rule:

An actor can be executed when tokens are present on each of its input arcs.

In the static dataflow model, a *one-token-per-arc* restriction and a *non strict operator* behavior are incorporated by extending the firing rule as such:

1. An actor ready to fire, can actually fire, if its output arcs have no token on them.
2. Certain actors can fire, even when some input tokens are missing. For example, the merge operator in Figure 3.2, if the boolean value of its control token is True, can fire if there is a token on the data input arc at the True side, even if at the False side there is none.

3.2.1 Data Structures as DAGs

In [31], in order to deal with complex data structures, Dennis introduced a *heap*, which is a *directed acyclic graph* (DAG), or simply a *tree*. The motivation behind this approach, was to avoid excessive copying by sharing common substructures between structures. Tokens carry a pointer to nodes of the tree and not the whole tree itself. Associated with every node, there is a reference count indicating how many tokens point to that node.

The domain of values for the dataflow programs will include *elementary* and *structured* values. The set of elementary values will be *boolean* $\bigcup$ *integers* $\bigcup$ *reals* $\bigcup$ *strings*. The set of structured values contains all finite sets of *selector-value* pairs such as:

$$[< s_1 : v_1 >, ..., < s_k : v_k >]$$

where s_i are distinct elements in *integers* $\bigcup$ *strings* and v_i are either elementary or structured values.

The tree may have many roots and is structured in such a way, that there is a directed path to every node from at least one root. The edges of the tree are labeled as selectors, with the restriction that two edges branching out of a node, cannot have the same selector.

The nodes are either *elementary nodes* or *structured nodes*. As suggested by its name, an elementary node represents an elementary value and has no emanating edges. A structured node represents a structured value, where each selector s_i is the label of one of the emanating edges and v_i is the value of the node, to which that edge leads. A root can be of either type.

For example, consider this representation of the following 3×3 array:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

As a structured value it will be represented as such:

$$\begin{aligned} [< s_1 : \ [< s_{11} : 1 >, < s_{12} : 2 >, < s_{13} : 3 >] \ >, \\ < s_2 : \ [< s_{21} : 4 >, < s_{22} : 5 >, < s_{23} : 6 >] \ >, \\ < s_3 : \ [< s_{31} : 7 >, < s_{32} : 8 >, < s_{33} : 9 >] \ >] \end{aligned}$$

Its DAG representation, as shown in Figure 3.4, would be a two-level tree, whose interior vertices have 3 children and its leaves store the values of the array.

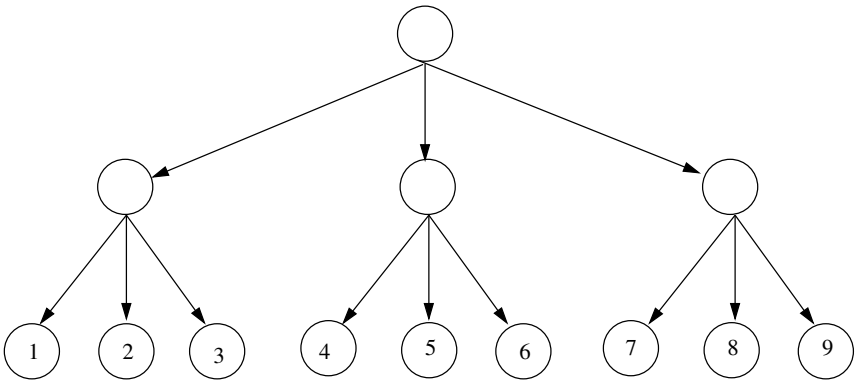

Figure 3.4: The Heap Of An Array

Copying the array is straightforward. We can create a *copy* actor, which creates a token pointing to the same node of the heap (in this case the root of the tree) as the input token. The reference count of the root is incremented to reflect the additional reference to it.

While this approach deals with the problems discussed in the previous chapter, it has the major drawback, that only one modification can occur at a time. Thus, for example, the modification of different cells of an array, which could be done in parallel, is sequentialized.

3.2.2 An Architecture for Static Dataflow

To provide further insight on the dataflow model, it is useful to understand how hardware, that can support this computational model, could be designed and its differences from the

conventional von Neumann machines. In this chapter, we discuss, how the architecture Dennis presented for the static dataflow model [32], deals with the problems, faced when trying to implement the exact abstract model.

Initially, the dataflow schema to be executed is stored in memory. Also, with the one-token-per-arc restriction, storage for tokens can be allocated prior to execution, since the number of edges is fixed for any program graph. The memory is organized into *Instruction Cells*, each one corresponding to an operator of the program graph. A cell is an expansion of the basic *instruction*, which includes the operands (data packets) for the execution of the instruction. An instruction specifies the operation to be performed and the addresses of the registers, to which the result must be directed (Figure 3.5).

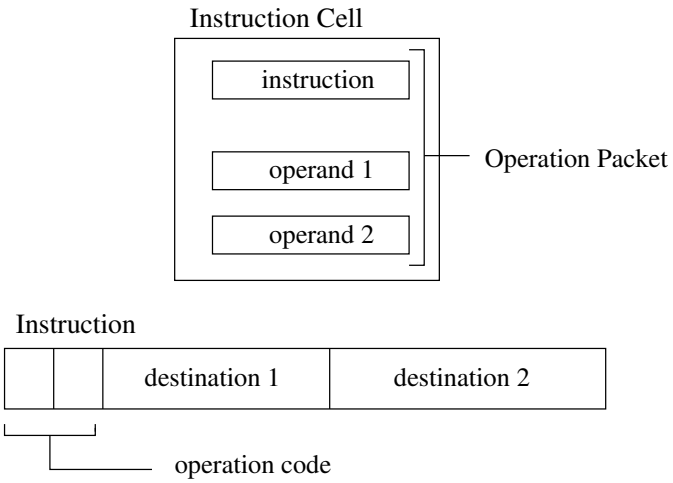

Figure 3.5: Basic Instruction Format For A Static Dataflow Architecture

The cells have *presence flags* to indicate the existence of a stored value, so it is straightforward to determine if the inputs for an operation are present. Addresses of *enabled instructions* (instructions ready for execution) are stored in an *instruction queue*. A *fetch unit* removes the first entry, collects the corresponding data, forms an operation packet and sends it to an available *operation unit*. The operation units operate concurrently. Each one computes a result and generates a result package for each destination.

The one-token-per-arc restriction cannot be implemented at the hardware level. Rather, the program graph is transformed to contain *acknowledgement arcs*. A token in an acknowledgement arc indicates that the corresponding data arc is empty. A node can fire if it has tokens on all data and acknowledgement arcs.

3.2.3 Discussion

The static dataflow architecture's main strength lies in its simplicity. It is easy to determine if the input tokens are present and since an arc can hold at most one token, memory can be allocated at compile-time, thus eliminating the need of creating complex hardware to manage the tokens.

However, this model has some severe problems. The additional acknowledgement arcs can substantially increase the token traffic by a factor of 1.5 to 2 [4]. Moreover, while it can exploit structural and pipelined parallelism, it can't benefit from dynamic forms of parallelism, such as *loop parallelism*, i.e. dynamic unfolding of loops. For instance, consider the program graph in Figure 3.2, being used in a loop. To abide by the one-token-per-arc

restriction, only one evaluation of this graph could be carried out at a time, since the input arc for c would have a token, till the division operator at the end consumed it.

3.3 Dynamic Dataflow

The *dynamic* or *tagged-token dataflow* model is an alternative approach, proposed independently by Watson and Gurd [76] and Arvind and Culler [4]. In this model, a *tag* is associated with each token, uniquely labeling it as it is dynamically generated during execution, specifying its conceptual order within a stream of tokens on an arc. This enables us to maintain a logical FIFO order, freeing us from the limitations of the actual physical arrival order of the tokens.

3.3.1 Motivation

Let's remember the program graph of Figure 3.3, which computes the sum $\sum_{i=1}^{N} F(i)$. Also, let's suppose that the "blob" representing the function F, is a complex graph, which needs a long time to execute, relative to $+$.

The static dataflow model implements a one-token-per-arc restriction, which substantially limits the parallelism that can be exploited in the program. Assuming a capacity of one token on an arc, F will soon block the switch operator, which won't be able to fire, after having already produced a token on its arc directed to F. In turn, the switch operator will block the merge operator and so on. Clearly, while F is executing, not much will happen in the rest program.

Now assume an unbounded token capacity on arcs, greatly improving the parallelism exploited and the performance of the program. Incrementing the index variable i and evaluating the predicate $\leq N$, do not depend on F. If F executes much slower than the rest operators, tokens will pile up on its input arc, waiting to initiate a new execution. In the tagged dataflow model, as soon as a token reaches F, it can initiate a new computational activity.

Note that a subtle problem arises. Will the result be the same, if the second invocation of F completes before the first? In our example, since addition is an associative operation, it is obvious. We will see, that the dynamic dataflow model maintains determinacy.

3.3.2 Tags

We will describe the classic dynamic model, using the constructs and notations of Arvind and Gostelow [8], also presented in [4].

First, we will need the meanings of activities and code blocks. An *activity* is a single execution of an operator. A *code-block* is a graph, that is either acyclic or a single loop. A program can be viewed as a collection of code-blocks, where each node is identified by a pair <*code-block, instruction address*>.

We prescribe a unique name to each activity generated and each token carries the name of its destination activity (tag). A tag consists of four fields, *u*, *c*, *s* and *i*:

- *u* is the context field, which identifies the context in which a code-block is invoked. Concretely, it distinguishes between between different invocations of a code-block. Note that the definition is recursive and the context field is itself a tag.
- *c* is the code-block name. Every code-block has a unique name.
- *s* is the instruction address within the code-block. Remember that the instruction address and code-block name identify a node.
- *i* is the initiation number, which distinguishes between different iterations of the same invocation of a loop code-block. If the activity occurs outside a loop, this field is 1.

So a token is represented as such: $<$*u.c.s.i, data*$>$. In case the destination operator requires more than one input, the token is represented as $<$*u.c.s.i, data*$>_p$, where p is the port number (i.e. it specifies to which input arc it belongs).

3.3.3 Tagging Rules

The way these unique activity names are generated (*tagging rules*) must allow for *user defined functions*, *conditional*, *loop* and *recursive* constructs. Moreover, it is obvious that this mechanism should create the names in a distributed manner.

Functions and Predicates. Assume an instruction s in a code-block c (identifying the node <c, s>), that performs a binary function f, receives a token with data x on its first input arc, a token with data y on its second input arc and the result's destination is instruction t. The tagging rule is:

$$<u.c.s.i,\, x>_1 \times <u.c.s.i,\, y>_2 \;\Rightarrow\; <u.c.t.i,\, f(x,y)>_p$$

Conditionals. Assume a *switch* operator receives a token with data x on its data input arc and a control token with boolean value b. The tagging rule is:

$$<u.c.s.i,\, x>_{data} \times <u.c.s.i,\, b>_{control} \;\Rightarrow\; \begin{cases} <u.c.s_T.i,\, x>, & if\ b = true \\ <u.c.s_F.i,\, x>, & if\ b = false \\ undefined, & otherwise \end{cases}$$

No two waves of input tokens carry the same u and i and for any given tag, exactly one of the successor instructions s_T, s_F will receive a token carrying it. Also, since we can maintain a logical FIFO order and ignore the actual physical one, there is no longer a need for the *merge operator*; the token streams produced at each output arc of the *switch* operator can be merged in an arbitrary fashion.

Thus, continuing the example from [4], the conditional graph from Figure 3.2 transforms into the one in Figure 3.6, where the black circle is not an operator; it denotes the convergence of two arcs on one.

Loops. Lets remember the loop graph in Figure 3.3. To implement the loop, a *switch* operator was used, which routed the tokens back into the body of the loop, while the boolean value of the control token was true. To implement loops in the dynamic model, we require four more operators: *D* and D^{-1}, *L* and L^{-1}. These operators only affect the

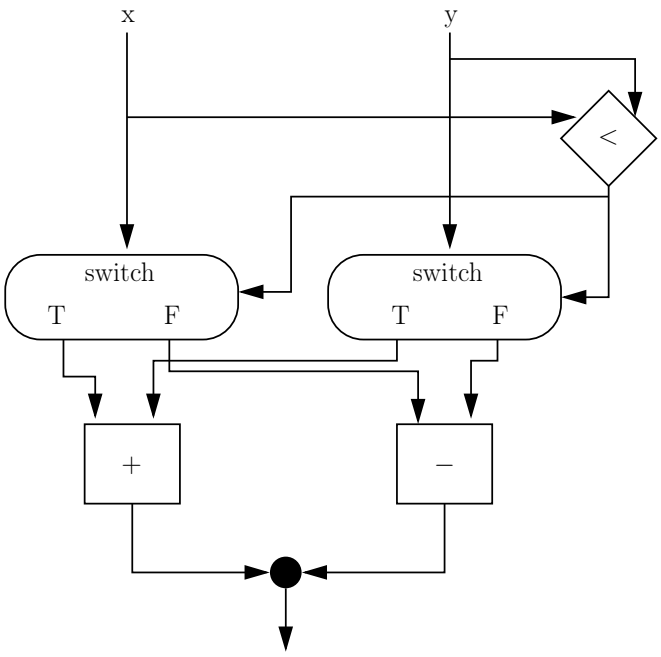

Figure 3.6: A Conditional Graph In Tagged Dataflow

tag of the token passing through, and not the data itself, so they can be viewed as control operators.

The *L* operator creates a new context $u' = (u.c.s.i)$ and a new code-block name c' for every new instantiation of a loop. This is necessary in case of nested loops, where there can be many active invocations of the inner loop. The tagging rule for *L*:

$$<u.c.s.i,\, x> \;\Rightarrow\; <u'.c'.t'.1,\, x>$$

The *D* operator increments the initiation number after each iteration:

$$<u'.c'.t.j,\, x> \;\Rightarrow\; <u'.c'.t'.j+1,\, x>$$

When the *switch* operator sends a token with boolean value false and ends the iterations, the D^{-1} operator sets the initiation number to 1:

$$<u'.c'.w.n,\, x> \;\Rightarrow\; <u'.c'.w'.1,\, x>$$

The L^{-1} operator can be considered the reverse of *L*. It unstacks the context stacked by the corresponding *L* and restores the initial context, code-block name and initiation number:

$$<u'.c'.w'.1,\, x> \;\Rightarrow\; <u.c.s'.i,\, x>$$

The loop graph from Figure 3.3 transforms into the one in Figure 3.7.

Procedure Application. To implement procedure application we make use of two operators, *A* and A^{-1}. Moreover, each procedure is prefixed by a *BEGIN* operator and suffixed by an *END* operator.

The operator *A* takes two inputs, the code-block name q of the procedure to be applied, an argument a to pass to the procedure and creates a new context for the new procedure:

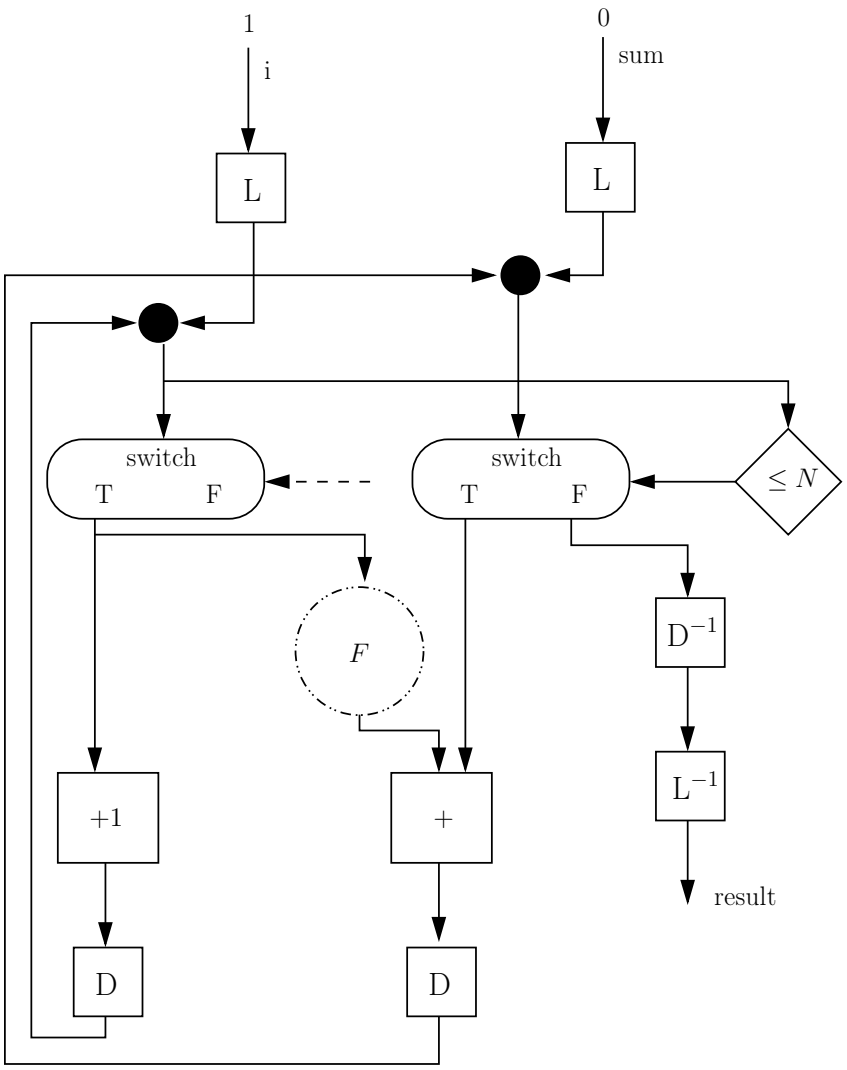

Figure 3.7: A Loop Graph In Tagged Dataflow

$$<u.c.s_A.i,\ q>_{proc} \times <u.c.s_A.i,\ a>_{arg} \ \Rightarrow\ <u'.c_q.begin.1,\ a>$$

where $u' = (u.c.s_T.i)$ and s_T the address of the A^{-1} operator corresponding to the A in s_A.

The *BEGIN* operator replicates the token for all its emanating arcs. The *END* operator restores the initial context stacked by the *A* operator and returns the result:

$$<u'.c_q.end.1,\ b> \ \Rightarrow\ <u.c.s_T.i,\ b>$$

where $u' = (u.c.s_T.i)$. The A^{-1}, like the *BEGIN* operator, merely replicates the result for all its successors.

3.3.4 Incremental Data Structures

The discussion for complex data structures in dataflow till now, indicates that parallel data structures are an awkward and anything but trivial matter. A common approach in the dynamic model is the use of *I-structures* (for *incremental structures*) [7, 6, 4]. As the name suggests, the structures are constructed incrementally, in a *monotonic* fashion. Semantically, we can view I-structures as a logic variable abstraction [67].

From the programmer's viewpoint, an I-structure is an array of slots. It's a *non-strict data structure*, meaning we can use and modify it, before it is fully defined. To achieve this, we implement a policy of *deferred reads*; if a read request, for a slot not yet defined, arrives, the system defers the request, till the time it can be satisfied. With some ad hoc hardware for the implementation of queues of deferred reads, associated with every cell, deferred reads can be processed quickly. Thus, we don't lose any parallelism, as pointed out in Subchapter 3.1.3.

As in the approach to structures in the static dataflow model, the token carries a pointer to the structure. All I-structures reside in global memory and read/write operations are handle by an abstraction module, the *I-structure store*. Conceptually, all I-structures belong to it.

Selection. Every "read token" contains in its tag the address of the component of the I-structure to be selected (or read). When it arrives, it creates an *I-fetch* request to the I-structures store. If the component specified in its tag is present, it is returned. Otherwise, the request is deferred, or more concretely, the tag is queued in an *deferred read requests* area.

Assignment. A request for an assignment to an I-structure becomes an *I-store* request to the I-structures store. If the corresponding location is empty, the value is stored and the location is marked as present. If there are any deferred reads, now they can be satisfied. If there already was a value at that location, a runtime error is generated, since a single-assignment rule is imposed; we can write to a location only one time.

3.3.5 Overhead and Excessive Parallelism

In Subchapter 3.3.1, we got a glimpse of the order of parallelism, which can be exploited even in a seemingly simple and sequential problem. Consider the following program, with nested loops:

```
for i=1 To n Do {
   for j=1 To m Do {
      A[i][j] = f(i+j);
   }
}
```

As usual, supposing a functional *f*, we can have $m \cdot n$ active computations of *f*, at least in theory. Now consider a program with even more levels of nested loops. It is unlikely, that an architecture will have enough PEs to run all these function invocations simultaneously.

Instead of the expected speed improvement thanks to the parallelism exposed, the piling of *waiting tokens* will slow down the execution and consume all memory. It is not hard, to even imagine scenarios, in which the system will deadlock; if tokens with unmatching tags flood the arcs, not leaving space for new tokens, the node will never find two tokens with matching tags, so as to compute a result.

Moreover, this model of *asynchronous parallel execution* incurs severe overhead [26]. When partitioning the instructions executed into *basic work* and *overhead*, such as instructions for manipulating tags, empirical data suggests that the overhead is comparable to the basic work. This overhead could diminish, if the parallelism exploited was limited.

Hence, we develop the notion of *useless parallelism*, in the sense that it cannot possibly be exploited. It is a trade-off for giving the programmer a powerful tool, to exploit parallelism

in an implicit manner, without reference to the actual hardware. It is necessary to limit this useless parallelism and save resources, without of course sacrificing performance.

One solution, is to limit the maximum number of concurrent iterations [27], through *loop bounding*. The way this is implemented at the MIT tagged dataflow machine [6], is based on the observation, that a new iteration begins after a token carrying a boolean value reaches the *switch* operator, which routes tokens in and eventually out of the loop body. By delaying those tokens, we can essentially hold back the iterations, imposing a ceiling. The number of allowed concurrent iterations can be specified either at compile-time, or dynamically, based on the load of the system.

3.3.6 Discussion

Everything about the dynamic model summarized till now, reflects the work of Arvind and his team. The tagging rules and I-structures as presented above, are the foundations of the MIT Tagged-Token Dataflow project. The interested reader can look at [6] to delve into an architecture for the dynamic model.

3.4 Synchronous Dataflow

Synchronous dataflow (*SDF*) is a subset of the pure dataflow model, not fully general as the latter. Despite the restrictions imposed to it, the optimizations that can be made and the decrease in overhead (due to the asynchronous nature of classic dataflow), make up for the sacrifice in expressivity. SDF is extensively used for digital signal processing and is generally a more appropriate candidate for real-time and reactive systems, than pure dataflow [14].

3.4.1 Synchronous Dataflow Graphs

In synchronous dataflow graphs, the number of tokens an actor consumes or produces when firing, is fixed and known at compile time. For example, the program at Figure 3.1 is a synchronous dataflow graph. In Figure 3.8, next to its emanating or incoming arc, there is the literal ”1”, indicating the number of tokens each actor consumes/produces upon each execution.

Since the number of tokens that are consumed/produced by an actor must be known at compile time, it is obvious that certain programs cannot be represented. Notice that the *switch* and *merge* operators are asynchronous. Depending on the control token’s boolean value, a *switch* node can produce zero or one token on its output arcs, and a *merge* node can consume zero or one token from its input arcs. Consequently, conditional graphs (eg. in Figure 3.3), are not SDF graphs. Moreover, if all nodes are restricted to be SDF, only loops with a known (at compile-time) number of cycles of the iteration, can be specified.

3.4.2 The Trade-Off for Expressiveness

Yet these limitations offer considerable advantages, such as compile time predictability.

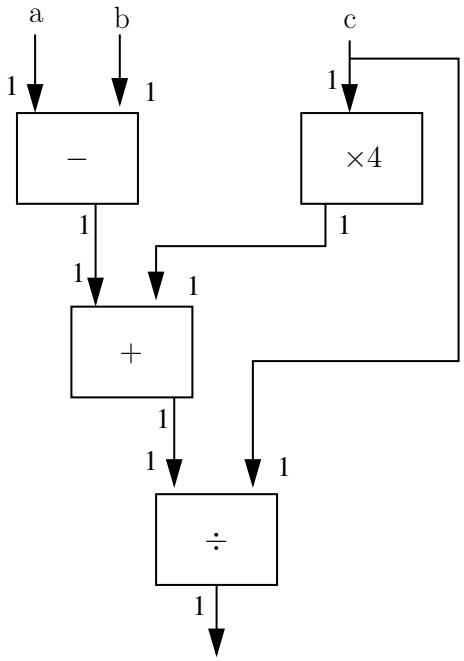

Figure 3.8: A Synchronous Dataflow Graph

The advantage of SDF is, that it can be statically scheduled [20, 53, 54]. Intuitively, we can form a *periodic schedule* and know how many times a node should be invoked at each cycle, so instead of a general dataflow graph, we can have an *acyclic precedence graph*.

By turning the graph into a sequential program (on one or multiple processors) and avoiding the dynamic scheduling, the runtime overhead evaporates. This addresses a major requirement of real-time systems; response-time.

It is worth mentioning, that even dataflow graphs which are not SDF, may contain subgraphs that are, and allow for partial static scheduling.

4. DEMAND DRIVEN DATAFLOW

In this chapter we will examine how dataflow programs, as described in Chapter 3, and particularly in the tagged dataflow model, can be computed in a demand driven manner, as described in Chapter 2.

The basic idea behind demand driven dataflow is as follows:

A node will fire, if there are sufficient tokens on its input arcs and there is a demand for its output.

Conceptually, the *demands* can be visualized as special tokens (*questons*), going against the flow; they go from the end of an edge to its beginning. When a node receives a demand for its output, it will itself generate demands for the input it requires and so on.

We can imagine this process as a branching tree. The root at the top, is the node that will yield the result of the program. The root generates demands for input tokens, which reach its children. Those children in their turn, will create more demands, further expanding the tree, as these demands travel downwards. Hopefully, this process will terminate and values will begin flowing back, up the tree. When they reach the root, we will have the final result.

This process was termed by Wadge *eduction*. Eduction's definition in the Oxford English Dictionary is:

The action of drawing forth, eliciting, or developing from a state of latent, rudimentary, or potential existence; the action of educing (principles, results of calculations) from the data.

We will first mention some problems or difficulties with data driven evaluation of dataflow graphs, which are better addressed with demand driven evaluation, and then introduce *operator nets*; dataflow networks, which are not evaluated using ”dataflow”, along with changed syntax and a bit more complicated semantics.

4.1 Motivation

4.1.1 Pointwise Nonstrict Operators

In Chapter 3, all the operators we examined had exactly one output arc, except for *switch*. We introduce another operator, which can give us the effect of *switch*, the *whenever* operator [10], or *wvr* for short.

The *wvr* operator has two input and one output arcs. One input arc is for the control token and the other for a data token. If the control token's boolean value is true, the data token is passed to the output arc, otherwise it is discarded. Notice that it is possible, that a *wvr* node may produce less results than the number of its input sets. The behavior of the *switch* operator can be mimicked, as shown in Figure 4.1.

An operator or function is *pointwise* if its i-th output depends at most on its i-th set of inputs. For example, simple addition is a pointwise operation, while the *wvr* node is not, since its first output may depend on its second set of inputs.

An operator or function is *pointwise strict*, if it is pointwise and if, in order to produce its i-th result, the i-th set of inputs must be fully defined. For example, the *merge* node in the

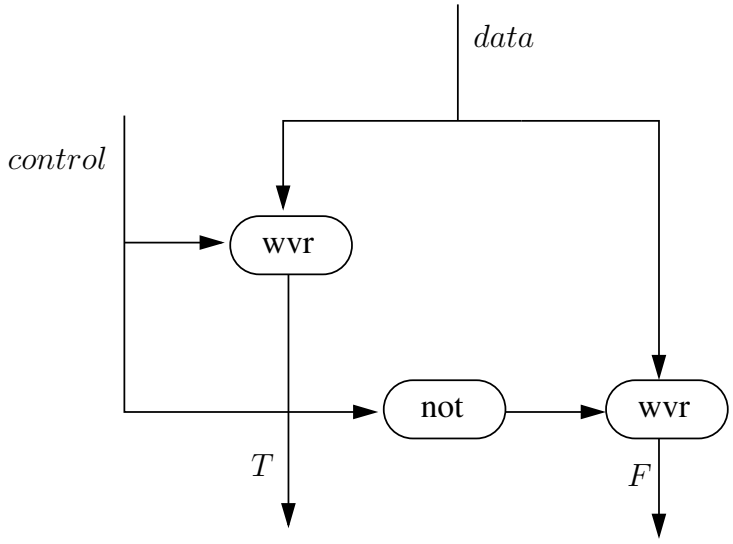

Figure 4.1: Breaking Down *switch* To *wvr*

static dataflow model is not pointwise strict, since if the control token's value is true and there is a token only on the data arc at the true side, the node will still fire.

Now lets turn our attention to the following simple expression:

if B **then** E_1 **else** E_2

There are two ways to evaluate it, as in Figure 4.2. In the left network, first we evaluate the boolean expression *B* and then compute either E_1 or E_2. In the second, we compute both E_1 and E_2 first, and then after evaluating *B*, we produce as result the appropriate value. The half bottom part of the second network, can be viewed as a ternary pointwise nonstrict *if-then-else* operator. At most two of its input arcs will have input; the control arc and either the true or false data arc.

Clearly, the approach taken in the second network, should be avoided, since redundant computations are performed. In this example, it appears simple to reform the program, so it takes the form of the left one, and avoids the redundant computations. However, a general automatic transformation scheme to transform programs, so as to rid the unneeded computations, has not been found.

In languages, where we allow pointwise nonstrict operators, it is not clear how to efficiently compute the programs, using data driven computation.

4.1.2 Nonstrict Functions

Operationally, a nonstrict function is one, which may not evaluate all its arguments. The motivation is the same as in the previous subchapter, but the difficulties in implementing such behavior in a data driven organization, are not.

In data driven dataflow, defined functions are eagerly evaluated. *Eager evaluation* means, that the arguments are evaluated prior to the function call. And indeed, for a function to be called, there must be tokens on all its input arcs.

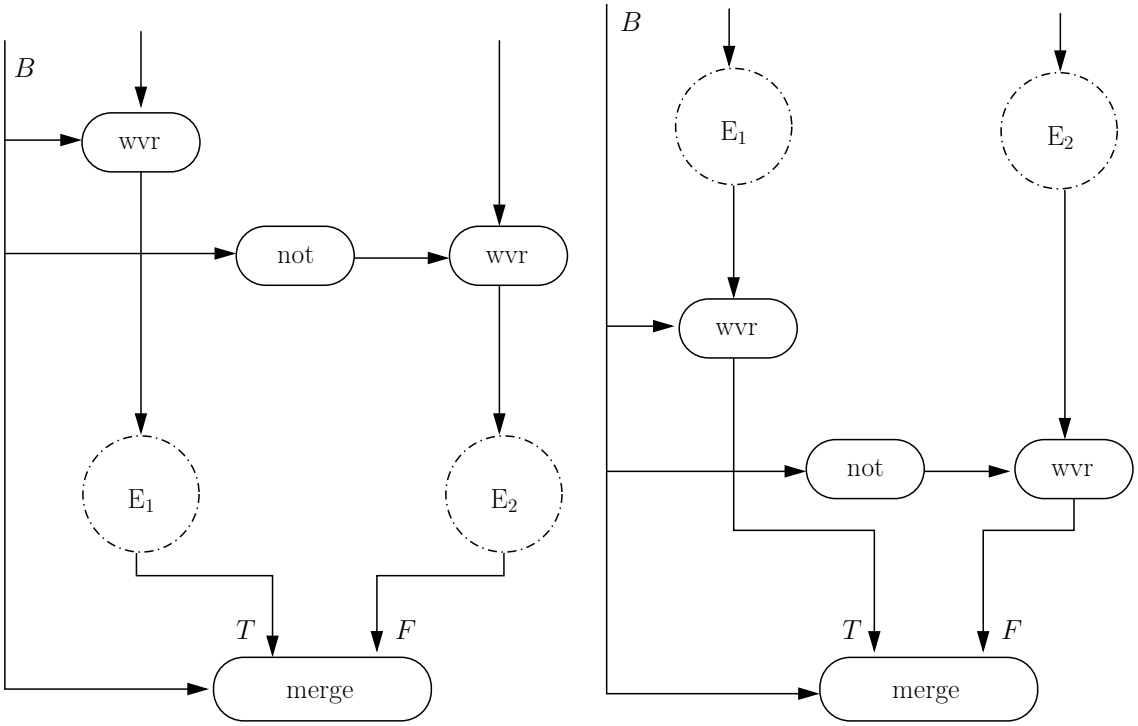

Figure 4.2: Two Evaluations Of *if-then-else*

Lazy evaluation or *call by need* entails that an argument will be evaluated only when it is needed. We saw, that a simple transformation of the program, can solve the problem of redundant operations in the case of an *if-then-else*, implicitly imposing a lazy evaluation. The difference to when trying to apply the same logic to function calls, is that an operator, upon receiving a token, knows if it can fire, even if some tokens are missing. This does not stand for a function call.

Without actually evaluating the function, we cannot know if we will need a particular argument. The only solution is to call the function, but delay the evaluation of arguments until they are needed. When a request for one is met, the argument can be evaluated. Of course, there would potentially be a need for further tag manipulation operators, to send its argument to the right invocation of the function, introducing further overhead.

The consensus is, that although these problems are in theory solvable, it is not worth it.

4.1.3 Nonstrict Data Structures

This is not the place to dwell in the importance of nonstrict structures; let's assume we want that functionality! The following example, entailing *I-structures* [7] and eager evaluation, demonstrates the awkwardness of nonstrictness in a data driven computation organization.

When a read request for a particular component of an array arrives, if that component is not yet computed, the read request is deferred, till the time it can be serviced. This behavior is unnaturally implemented in the data driven evaluation paradigm; dynamic synchronization techniques and extra storage for the deferred requests are needed.

In lazy evaluation, this synchronization problem can be solved in a more subtle way. Upon allocation, each component could hold a *suspension* for its computation, and a read re-

quest upon arriving, would force its evaluation. The testing of a reference to check if it is a suspension or a value in *graph reduction*, naturally translates into a synchronization test in dataflow's context.

Moreover, using lazy evaluation, we can handle *infinite structures*. An example taken from [75]:

```
X = 1 : : X
```

X is a list whose head is the number 1, and its tail is itself. Anyone unfamiliar with declarative programming and lazy evaluation, would be quick to dismiss this program as nonsense. *X*'s meaning could be either $\perp$ (operationally thought of as non termination), or the infinite list [1,1,1,1,1,1,...]. In the case of the latter, the eager evaluation approach crambles; it makes no sense to try to eagerly compute an infinite list. However, in demand driven evaluation, it is possible to compute a partial list - exactly the part that is actually needed.

4.1.4 Management Of Resources

In tagged dataflow, the excessive asynchronous parallelism can quickly lead to saturation of resources [27, 26]. Although perhaps a by-product, a demand-driven evaluation, can implicitly restrain this need for resources, further from the obvious gain from avoiding redundant computations.

Consider the computation of $\sum_{i=0}^{n}[f(x) + g(x)]$, and suppose that f takes much less time to compute than g. What will happen in the data driven evaluation, is that the tokens generated by f will flood the system, piling on one arc of the $+$ operator, while the tokens from g will arrive infrequently. This not only puts unnecessary strain on the system in regard to storage, but it also keeps the PEs busy with more invocations of f, while they could compute something else.

In demand driven evaluation, demands would implicitly prioritize computations, helping compute what is immediately needed and avoiding the piling of packets with premature values.

4.2 Operator Nets

The language of *operator nets* is a simple graphical language, which is mainly uninterpreted; different (continuous) data algebras and sets of input provide mathematical semantics to it [10], giving birth to new particular languages, of different degrees of expressibility. Consequently, the search for a dataflow language, roughly translates to the choice, of which sequence algebra will interpret the operator nets.

Operator nets can be viewed as a generalization of the *"simple language for parallel processing"* defined by Kahn [50], and denotational semantics can be given in a similar way.

The main differences from dataflow graphs presented so far are:

1. Operator nets can be evaluated either in a data driven, or in a demand driven fashion.

2. Every operator net denotes a function - a net is a composition of functions, corresponding to its subnets.

4.2.1 Syntax

An operator net is a *main* directed graph, possibly with named *subsidiary* operator nets, which are used as *function definitions*. Moreover, a node can be a *subcomputation* node, which corresponds to a particular subsidiary net.

All nonfork nodes have one output edge. Function and subcomputation nodes have the same number of input edges as their arity, which is the number of input ports of the corresponding net. Note that the recursive definition of operator nets, allows functions definitions within function definitions, thus recursive function definitions.

As an example, consider the following program that sums the squares of two numbers, written in Lucid [75], which is itself based on operator nets. The corresponding net is depicted in Figure 4.3. The *square* node, is a subcomputation node, corresponding to the subsidiary net named *"square"*.

```
square(x) + square(y)
where
    square(x) = x*x;
end
```

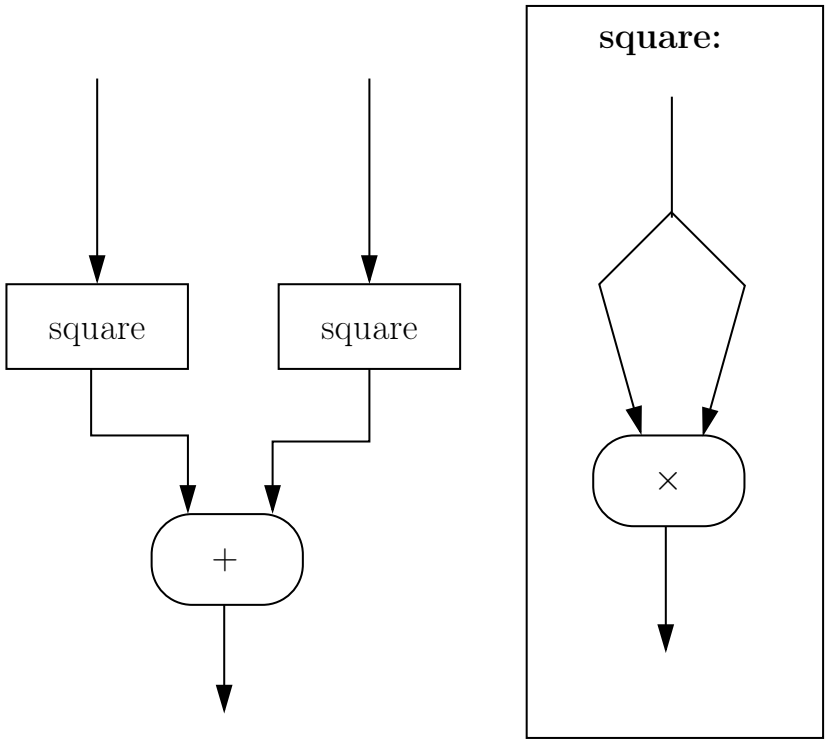

Figure 4.3: A Simple Operator Net

4.2.2 Denotational Semantics

Arvind and Gostelow provided both denotational and operational semantics. Data driven and demand driven evaluation can be considered two different operational ways of achieving the same mathematical meaning. Dwelling into semantics is outside the scope of this thesis - only some properties will be mentioned, in order to understand some functionalities of operator nets.

An operator net can be reduced to a set of *equations*, with every subcomputation node containing the equations of the corresponding subsidiary net. For example, in Figure 4.4 the net of Figure 4.3 is depicted with the edges labeled, and these are its corresponding equations:

$$
\begin{aligned}
& r = +(d, e) \\
& d = square(x) \\
& e = square(y) \\
& square(a) = h \\
& \quad [\ h = *(a_1, a_2) \\
& \quad\ \ a_1 = a \\
& \quad\ \ a_2 = a\ \ ]
\end{aligned}
$$

Although the rules for creating these equations are omitted, the reader should intuitively be convinced that they ”represent” the operator net in question.

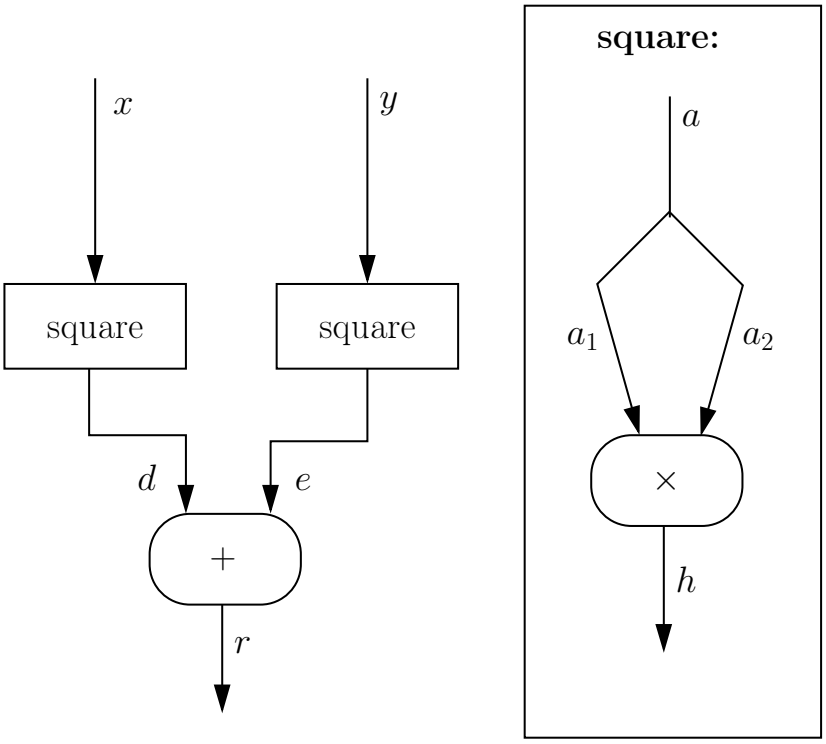

Figure 4.4: A Simple Operator Net With Labeled Edges

The meaning of the net is the solution of these equations, using standard *fixedpoint* theory, as in the work of Kahn [50]. Note that such a set of equations, can itself be considered a

language; for a particular setting of $E(A)$, the language is essentially *Lucid(A)* [75], which is the member of the Lucid family corresponding to the data algebra A.

Algebras. Assume an algebra of elementary objects (*data algebra*) A. For example, the universe of A can contain integers, booleans, etc. In addition, we add in its universe a *least element*, denoted by $\perp$ (*bottom*). The operational meaning of bottom is *nontermination*.

A function or operator f is called *strict*, if $f(\perp) = \perp$. The operators of A can be nonstrict, that is, not all its arguments need be defined ($\perp$) for its output to be defined. An example is the ternary nonstrict *if-then-else* operator, we discussed about in 4.1.1.

$I(A)$ is defined as the algebra of *infinite sequences* of elements of the universe of A. For example, the new least element will be the infinite sequence $\perp, \perp, \perp, ...$ and 3 will be $3, 3, 3, ...$. Notice that a sequence of $I(A)$ can be *intermittent*; $\perp$ elements can precede non $\perp$ elements, e.g. $1, 2, 3, \perp, 4, ...$. The operators of $I(A)$ are pointwise extensions of the operators of A.

The algebra $E(A)$ is an *enlargement* of $I(A)$, in which special *nonpointwise* operators are added, such as *first*, *next*, *followed-by* (*fby*), *whenever* (*wvr*), and others. These operators, are not pointwise strict; for a specific point i, they don't require all their arguments to be defined at that i point. But most of them are strict, with exceptions such as *merge*. Concretely, if an operator's argument is formed only by $\perp$'s, so will be its result.

4.2.3 Operational Semantics and Eduction

To give operational semantics, we assume operator nets behave as usual dataflow graphs (although there is no one to one correspondence); tokens flow through edges and into nodes. When a result is computed, it is pushed to the output edge. We restrict that the history of tokens passing through an edge, corresponds to the sequence, that is the value of the variable associated with that edge, when creating the equations of the net.

For different algebras, the mathematical meaning corresponds to piped or tagged evaluation. In Chapter 3, tags were introduced, as a means for increased parallelism. Another gain is, that tagged evaluation produces more results than piped evaluation. That is because, intuitively, in tagged evaluation we can "look ahead" in a stream. Moreover, using tags, copies of subsidiary nets need not be made; the tags distinguish between different invocations.

In data driven evaluation, operator nets behave essentially the same as dataflow graphs. What is of interest, is the (tagged) demand driven evaluation, and particularly the way these demands are propagated.

In the beginning of this chapter, we talked about how demands can be visualized as *questons* traveling back the network, from the end of an edge to its beginning.

The queston propagation for pointwise nonstrict and nonpointwise operators, does not follow some general rules; it depends on the operator itself. For example, for the ternary pointwise nonstrict *if-the-else* operator, the propagation of demands is as follows: When a demand for its value, arrives at the *if-then-else* operator, it creates a queston with the same tag as the one it received, and sends it back to its test edge. When the answer arrives, according to its boolean value, it creates yet another queston with the same tag, and sends it to the *then* or *else* edge respectively.

Lazy evaluation of functions, corresponds to the way questons propagate through sub-

computation nodes. When a demand for the output of such a node arrives, the node is replaced by the corresponding net. A queston, which keeps in its tag the ”time” this sub-computation was invoked, is used as a demand for the output of this net. If the net creates any demands for its input (the function’s arguments), the tags help match the values to the corresponding activation.

4.3 Combining Data Driven and Demand Driven

We have seen how demand driven evaluation rids the execution of redundant computations, potentially improving the execution time, as well as offering more results, by lazy function calling. However it is not without drawbacks of its own.

There have been attempts to combine the benefits of data driven and demand driven evaluation.

4.3.1 Problems with Demand Driven Evaluation

First, the propagation of demands is expensive. There are cases, in which this additional cost is a good trade-off, for avoiding a large quantity of superfluous computations. But in other cases, eg. when computing the product of two matrices, where no (or few) redundant computations would be made in data driven evaluation, the additional work of demands propagation will only delay the execution.

Moreover, the demand driven evaluation could be said to ”beat the purpose” of dataflow. Dataflow’s main appeal is, that it does not overspecify the sequence of computations, allowing massive parallelism to be naturally and implicitly expressed. By altering the firing rule, so nodes have to wait not only for sufficient input, but for a queston as well, the exposed parallelism is restrained.

For example, consider the producer - consumer relationship. Normally, in dataflow, the producer would begin producing before the consumer demanded so. By waiting for a demand to begin, potential parallelism is lost. One countermeasure for such pathological situations, would be *anticipatory evaluation* [46]. Since we expect that the consumer will eventually demand values for the producer, the producer can produce some results in advance.

Yet another technique, which offers good empirical results, is *speculative evaluation* [46]. That is, values that may or may not be needed, can also be computed in advance, in an eager manner. However, a plafond should be imposed on these eager computations, so as to minimize the potential superfluous work.

4.3.2 Eazyflow

Eazyflow is a hybrid model of evaluation [10], combining demand driven and data driven evaluation. The name comes from ”EAger and laZY dataFLOW”.

This model essentially divides the edges of an operator net, between lazy and eager edges. Conceptually, it creates a coloring of edges; edges on which the flowing values are generated eagerly (data driven) are colored green, while the rest are colored red.

The color of the edges determine the evaluation style. If the output edge of a node is green, that node will be data driven - it can fire if there are sufficient tokens on its input arcs. If its red, the node will be demand driven - it will need both sufficient input and a demand for its result, in order to fire.

In this model, both *speculative* and *anticipatory evaluation* are employed to increase efficiency. To avoid excessive superfluous computations, a bound to the results an node can eagerly produces is imposed. Moreover, care must be taken, to avoid the piling of waiting tokens; if an eager node produces many tokens and sends them on the input arc of a lazy node, those tokens will idly wait, till a demand arrives.

4.3.3 From Demand Driven to Data Driven

Another approach to efficient demand driven evaluation, surges from a doctrine advocating, that data driven can subsume demand driven evaluation.

The idea is, given a program P, transform it to LP, with LP having the property that, its data driven evaluation will perform no more computations, than the demand driven evaluation of program P. Pingali and Arvind introduced such a scheme [66], in which the original graph is augmented with ”demand propagation code”, which mimics demand propagation. However, this scheme is not general enough; eg, it does not support recursive functions.

5. HYBRID DATAFLOW

5.1 Introduction

5.1.1 Problems with Von Neumann Architectures

The von Neumann organization is widely spread, since its sequential nature makes it easy to understand and implement, and there have been many optimizations, at hardware level, to address some of its shortcomings [12], such as *latencies* due to memory requests. However these optimizations concern the case of uniprocessors. When it comes to multiprocessing, these problems, intrinsic to control-flow, change qualitatively.

There are two major difficulties in multiprocessing in control-flow [5]:

- **Latency.** Latency is the elapsed time, between making a request and receiving the associated response, during which the processor waits idly. A common strategy to remedy this situation, is to take advantage of *spatial locality*. For example, compilers can deduce which variables will be used more frequently and store their values in registers. However, this strategy collapses, if such a variable is shared between two processes, and the other process does not have access to its latest value. It appears that such optimizations cannot be readily transfered to multiprocessing.
- **Synchronization.** Exploiting parallelism in a program, essentially amounts to decomposing it to fragments, which communicate amongst them, transferring to each other the necessary data. Thus emerges the problem, of a potential read-write race. It is the programmer's duty to circumvent this problem; eg. all reads can commence after the last write. Moreover, the programmer should be mindful of the granularity of the tasks; making them too coarse, will probably restrain parallelism, while making them too fine-grained, will incur a significant cost in the form of *context-switching*, ie. assigning a different process to the processor and storing/loading the appropriate values.

5.1.2 Problems with Pure Dataflow Architectures

The *fine-grained* (or pure) dataflow architectures we have examined this far, although quite promising and despite extensive theoretical research, failed to prove themselves competitive with contemporary von Neumann architectures.

One glaring problem with both the static and dynamic dataflow, is the poor performance with sequential code. This is because, an instruction can be issued to the dataflow pipeline, only after the completion of its predecessor. Consequently, if there is not enough parallelism to exploit, the processor will be underutilized.

A further drawback in the tagged token architecture, is the overhead associated with token management. The overhead in space consists of the waiting tokens piling on the input arcs of operators (see 3.3.5). The overhead in time is due to context-switching after each instruction. Concretely, the intra-procedure communication is excessive; matching tokens are created for all instructions of the body, which is unlikely necessary. Moreover, since the synchronization occurs at the instruction level, the use of registers to benefit from locality is impossible.

5.2 Combining Control-Flow with Dataflow

Realizing that dataflow and von Neumann are not orthogonal, there have been many attempts to incorporate the benefits of both control-flow and data-flow into a new model. The intuition is to handle parallelism as in dataflow and to deploy control-flow techniques to handle sequential code and optimizations .

Imagine a spectrum, with dataflow at one end and von Neumann at the other. Closest to pure dataflow is *threaded dataflow*, *large-grain dataflow* and moving closer to the von Neumann style, *RISC dataflow*. There are more hybrid models, but mentioning them or explaining in detail the aforementioned, requires a lot of focus to the underlying hardware, which is outside the scope of this thesis. The interested reader, can look up more information here [3, 45, 52, 71, 74].

5.2.1 Threaded Dataflow

This approach confronts the problem of sequential code. In particular, subgraphs with low degree of parallelism are identified and transformed into a *sequential thread* of instructions. Such a stream of instructions is processed in succeeding machine cycles, without matching tokens (with the exception of the first instruction of the thread). Moreover, when executing instructions of a thread, registers are used to store intermediate results, effectively taking us away from pure dataflow.

5.2.2 Large-grain Dataflow

Large-grain dataflow or (*coarse grain dataflow*) moves yet another step further from pure dataflow. This approach is reminiscent of Kahn networks.

The idea is, to increase the granularity of subcomputations from a single operand to a chain of operations [16, 42, 55]. So, for example, two nodes which are connected by an arc, namely they have a data dependency between them and cannot operate simultaneously, can be assigned on the same PE, on the same sequential code segment.

There are *macro dataflow actors*, behaving as normal nodes do in pure dataflow, but each of these actors represents a sequence of instructions, executed in von Neumann style. An actor fires when all data dependencies for its first instruction are met.

5.2.3 RISC Dataflow

RISC dataflow is another hybrid model, which was intended to ease the transition from imperative to dataflow programming languages and allowed the execution of software written for von Neumann processors.

The main characteristics are a RISC like instruction set, modification of the architecture to allow multithreaded computations, addition of instructions to manage multiple threads and implementation of the global storage as I-Structures [63].

6. DATAFLOW AND PROGRAMMING LANGUAGES

6.1 Introduction

Programming languages can be seen as mappings between algorithms and computing models. A language that would map algorithms to the dataflow model, should be able to *implicitly express all the possible parallelism* of the algorithm and abide by the dataflow principles.

The languages of the *imperative class*, are not commonly thought suitable for the dataflow model [36]. Conventional imperative languages, are too tightly connected to the underlying von Neumann model. The explicit total order of computations, inherent to them, is obviously not suited to expose massive parallelism, or instructions scheduling based on data dependencies. Moreover, the von Neumann languages lack useful mathematic properties and simple semantics [12], such as reduction semantics, with no or very simple states, while denotational semantics for a dataflow language can be given with fixedpoint theory [30].

Earlier dataflow languages fitted the class of simple operational models [12], but later adopted a more *applicative programming style*, were given denotational semantics and useful mathematical properties (eg. for programs transformation and verification [11, 75]).

One cannot readily provide a definition, for what constitutes a dataflow language, since it overlaps with other classes of languages, especially with applicative languages. For example, Lucid at first was described as a functional language, and only later as a dataflow language. However, there is a consensus on some characteristics, or requirements of dataflow languages, initially posed by Ackerman [2]:

- freedom from side effects,
- locality of effects,
- equivalence between data dependencies and scheduling,
- single assignment rule,
- an unusual notation for iteration.

Many of these characteristics, intuitively, have the same purpose; tie the sequencing constraints to data dependencies.

Freedom from side effects. Even before the advent of dataflow languages, it was clear that any language meant for concurrent processing, should abide by this rule [72]. It is a necessary property, to ensure that sequencing constraints correspond to data dependencies. It can be broken in many ways, such as the use of global variables or procedures that alter their arguments. The dataflow model employs a strict solution; the use of *call-by-value*. A call-by-value procedure copies its arguments, rather than modify them.

Locality of effects. This means that instructions don't have unnecessary, far reaching dependencies. For example, assume a variable $temp$ in a large program, playing the role of a temporary variable. Also assume that it is once again used, in a different, unrelated segment of the code. While the logic of the problem may have allowed for concurrent

execution of these two segments, this variable introduces an unnecessary, false data-dependency. This problem can be simplified, by introducing the sense of *scope*, ie. define for a variable the area of the program, in which it is active.

Equivalence between data dependencies and scheduling. Concretely, all the information necessary for running the program (eg. sequence constraints), are available in the dataflow graph of the program.

Single assignment rule. Within the area of the program, in which a variable is active, it can be on the left side of an assignment expression only once. This rule offers clarity of code and ease of program verification. In addition, notice that when a variable is assigned a value only once, the order of instructions is not important.

An unusual notation for iteration. Iteration in dataflow languages can be tricky. There is a camp claiming that the principle of iteration is foreign to purely definitional languages, since the values of variables cannot be modified. Moreover, it becomes even more difficult when taking into account the single assignment rule. This dogma was brought down by a very elegant solution; using temporal operators to realize iteration [75].

6.2 Dataflow Languages

6.2.1 Lucid

Lucid, developed by Ashcroft and Wedge, initially was regarded as a functional language, meant to allow for formal proofs. However, through the choices of design and mathematical semantics, it became apparent that it was more suited to be classified as a dataflow language [75, 9].

Lucid is a definitional language, namely every statement is an equation. It comes with static, denotational semantics, but its creators advise programmers to think operationally or as if specifying a dataflow network. To see the depiction of a program written in Lucid as a dataflow graph, one can read the chapter 4.2 on operator nets. In essence, a Lucid program is an operator net, both in semantics and in behavior (it is evaluated as in Eduction).

Iteration in Lucid. Iteration in Lucid is realized through temporal operators (*fby* and *next*). The programmer can regard a set of statements (equations) as an iterative algorithm and the nullary variables as dynamic objects that denote different values at different times, or more simply as a stream. For instance, consider the following statement:

```
i = 1 fby i + 1;
```

The variable *i* can be regarded as a stream, with 1 at its beginning, followed by (*fby*) an infinite number of values, each one being the sum of its previous one plus 1, ie. 1, 2, 3, 4, ...

Increasing the dimensions. We saw that by regarding nullary variables as streams and through the use of temporal operators, iteration was realized. The question is, why only one dimension? Why not augment Lucid with any number of dimensions? Work on that path led eventually to the conception of Multidimensional Lucid, [11, 68, 13], an intentional language [69], which provides a most elegant and concise way to describe problems, express algorithms and write programs [35].

6.2.2 LUSTRE

LUSTRE is a synchronous dataflow language [22, 41]. Like Lucid, it is declarative, but much more similar to functional languages. It was mainly designed for programming reactive systems, since thinking as in the dataflow context could help programmers write programs, while easily visualizing them reacting instantaneously to external stimuli. Semantics were also given, and work has been done to provide a basis for program verification methodology.

6.2.3 GLU

GLU, standing for Granular Lucid, is a hybrid language based on Lucid and C [48]. To battle the inefficiencies of the pure dataflow model, as well as the difficulty of implementing Lucid effectively on conventional machines, GLU is better described as a large-grain dataflow language.

It maintains some of the intentionality of Lucid, but here the basic subcomputation unit is a user defined function and not a single instruction. These functions are written in C and are sequential code fragments. The parallelism between functions is implicitly expressed, as in Lucid.

6.3 Dataflow in Conventional Languages

The simple and intuitive way, to describe the parallel structure of a program and deal with continuous incoming data (streams), provided by the dataflow framework, did not escape the attention of even those unconvinced of the practicality of dataflow. There have been attempts to integrate or simulate some of its aspects in conventional languages, at least at a high, abstract level. Two glaring obstacles in efficiently implementing dataflow behavior in conventional languages, are the contrast between dataflow graphs and sequential code, as well as the practical issues of simulating dataflow in conventional machines.

6.3.1 Process Networks

An early attempt of integrating dataflow characteristics in conventional programming, was the implementation of process networks in Java [65]. As in Kahn networks, the communication between processes, which could run in parallel, was achieved through simulated streams, implemented as FIFO queues.

6.3.2 Streams

Trying to create an easy way for programmers to deal with continuously streaming data, Java 8 introduced streams and lambda expressions. The programmer can now write a segment of functional-style code, which can be run in parallel. The connection to dataflow languages is apparent: a stream can be perceived as a nullary variable in Lucid, and in order to safely run tasks in parallel, functionality is enforced.

There has been work to make Java Streams mainstream [57], as it appears to be a promising approach to various fields, as big data [23] and data mining [18]. Among the benefits

are the ability to scale and parallelize, the ease to deal with changing data and the easy programming style, which does not require the programmer to have particular knowledge on how to parallelize the code.

However, much work is still required, to deal with inefficient implementations, as for example in poor memory handling. But of course, the question of whether some problems are pathological and cannot be amended is always present, since we essentially try to mix dataflow principles and ideas with conventional languages and machines.

7. DATAFLOW AND DISTRIBUTED COMPUTING

7.1 Programming Distributed Applications

There are countless applications, with massive data sets (or *big data*) as input, that execution on clusters is a one-way street. However, programming such applications with conventional means, can be very expensive. Programmers must have specialized knowledge on developing parallel applications and on the hardware. Moreover, the programs themselves will most likely be very complex and hard to maintain or scale for different cluster.

There is a need for a framework, that will allow even programmers with little or no experience in developing parallel software to create applications what will be executed in a distributed manner. Moreover, the application should be scalable, meaning that running the software for a different number of computers, should not amount to rewriting most of the code. Also, the programmer should not have to take into account the capabilities of each computer. A good framework, would allow the separation of the computations themselves and the description of the parallel structure of the problem [38]. The system itself should decide how to run those computations and share the workload among workers.

There are many commercial systems for distributed computing. Watching their evolution, from the earliest ones, that with today's standards can only be thought as crude, to the newest ones, which come ever closer to an efficient ”good system”, it becomes apparent that they have started to abide by the doctrines pertaining to dataflow. Unfortunately, up to a point, these doctrines were being reinvented - it took some time till it was understood, that distributed computing could benefit from the already existent research on dataflow.

7.2 The Evolution of Distributed Computing Frameworks

MapReduce [29] was one of the earliest frameworks. Each computation is expressed as a series of jobs, consisted of two stages. The programmer merely needed to specify these stages, that is to write a *map* and a *reduce* function. *Map* took a pair of *key/value* and produced new ones, while *reduce* merged together values to produce a smaller set of values. The system itself was responsible for executing the jobs in parallel, but lacked expressivity.

Next step, was *MapReduce Online* [25], which allowed pipelining between jobs. This led to the support of continuous queries and to better performance and response time, something important as many applications dealt with real time data. In addition, it was possible to get some early estimates of the final result (*online aggregation*).

A milestone was the handling of iteration. In the original MapReduce, there could not be overlapping iterations; for the second round to begin, all jobs of the first round had to be finished. New systems like *Twister* [33], *HaLoop* [19] and *iMapReduce* [77] added caching mechanisms for data between loops and made the scheduler loop aware. iMapReduce also added persistent tasks, to avoid the overhead of creating new ones and increased the asynchronicity of Map tasks.

Another important step was towards the direction of incremental computations, needed in evolving datasets. This is a particular problem in database systems, which need to adapt to changes, without re-evaluating everything from scratch [40]. Similarly, frameworks to

deal with more efficient updates were devised. Two notable are *REX* [58] and *Incoop* [15].

However, all these systems use ad hoc solutions and strategies - there is no general framework in which to work and formalize the requirements and behavior.

7.3 Introducing Dataflow in Distributed Computing

There were early indications that dataflow could provide a generalized framework for distributed computing. In the dataflow model, parallelism is implicit, with the constant flow of data through the arcs, it would be easier to get partial results and research on distributed data structures, on iteration-level parallelism [17] and on recursion [39] already exists. In addition dataflow languages are suitable for distributed environments. For example, GLU or another large-grain dataflow language appears to be well suited for the job. The programmer does not go into details about parallelism, which is implicitly expressed and software written in such a language is obviously scalable. A model like eduction (in operator nets), could also provide a base for a task scheduler.

Even when looking at the early distributed computing frameworks, notions reminiscent of dataflow appear. For instance, one can find an analogy between the tasks and the vertices of a dataflow graph and among intraprocess messages and tokens traveling on the arcs. In particular, in some systems, like Dryad [44], an application is a dataflow graph and the task scheduling was based on it.

In [24], it was proposed that dynamic dataflow could serve as a formal framework for distributed computing systems. In the dataflow context, concepts such as streams and asynchronism are naturally perceived as data tokens flowing in channels and the firing rule, that a function is executed when its input is ready. Moreover, iteration in a distributed manner could be achieved as in the old dataflow systems, where tags carried information about the context, such as round of iteration. Additionally, an idea on integrating iteration with dataflow, as well as achieving incremental iteration, which is useful in many applications, was presented in [34].

7.4 Distributed Computing Systems Based on Dataflow

7.4.1 Ciel

Ciel [62] is an execution engine for distributed dataflow programs. It supports both iterative and recursive algorithms. It uses lazy evaluation and comes with its own language for expressing task-level parallelism, *Skywriting* [59].

7.4.2 Naiad

Naiad [60], is based on what is described as *timely dataflow* [61], where the vertices are stateful and the messages are logically timestamped, to differentiate through contexts. One can easily regard this as as a variation of the classic dynamic dataflow model. Instead of writing programs using directly the low level primitives of the timely dataflow abstraction, programmers can also use higher-level interfaces, such as SQL or MapReduce.

7.4.3 Differential Dataflow

Differential dataflow [56] is a model for incremental computations, which allows arbitrarily nested iteration and describes how incremented computations can be efficiently implemented in the context of a declarative dataflow-like language. This project was built on Naiad.

7.4.4 TensorFlow

TensorFlow [1] is a machine learning system that uses a single dataflow graph to represent all possible computations in an application. It spreads the vertices across a variety of machines in a cluster, while all data dependencies are expressed explicitly in the graph.

7.5 Coordination Languages and Distributed Computing

Until now, we focused on the need for powerful systems, which can cope with big data in a distributed manner, but we omitted the need for appropriate languages to express algorithms for such frameworks. It is the author's opinion, that attempts to develop such languages lag behind and that many ideas could be derived from macro dataflow languages, par excellence suitable for distributed environments.

There is a consensus about splitting the computations of the algorithm and the actual description of the parallel structure of the application, a principle by which all distributed computing frameworks abide. Moreover, a coordination language [38], which should be a part of every framework for distributed applications, need not offer more capabilities than the system itself can offer. So in the past, simple libraries on top of conventional languages, like C, sufficed. However, with the advent of the newer systems, we need more expressive tools.

A first step would be to shed the fear of declarative programming becoming a mainstream programming paradigm. In the declarative paradigm, since the programmer merely provides the declarations, separating the computations themselves from the scheduling is done automatically. Moreover, procedural languages, fail to provide a natural framework to deal with parallelism - the writing of code becomes too messy. For example, Skywriting [59], while better than some previous coordination languages, still lacks the naturalness a declarative language could provide. Languages such as *Pig Latin* [64, 37] obviously move towards that direction, but still try to keep the appearances of conventional languages. *Distributed SociaLite* [70] which is used for graph analysis programs, is a good example of where we should head.

8. CONCLUSIONS

The dataflow computational model has been the focus of extensive research the past decades. Even though, the von Neumann model dominated the market, it does not mean that dataflow cannot claim its own share in the future, as dataflow machines are still used. So, even more research on it, is required.

Furthermore, the von Neumann style does not seem to cope with distributed computing and is very awkward with parallel software. So, nowadays, that applications are becoming ever more complex and are called to deal with big data, the dataflow model can provide an elegant framework for distributed computing systems. And indeed, more and more systems are adopting its principles, both as an execution model and as an abstraction for programming languages. It is undoubtful, that in future, dataflow will pervade the field of big data even further.

ABBREVIATIONS - ACRONYMS

FIFO	First in, First out
DAG	Directed Acyclic Graph
PE	Processing Element
SDF	Synchronous Dataflow

REFERENCES

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., G. Murray, D., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., and Zhang, X. TensorFlow: A system for large-scale machine learning.

[2] Ackerman, W. B. Data Flow Languages. *Computer 15*, 2 (Feb. 1982), 15–25.

[3] ARVIND, and BROBST, S. THE EVOLUTION OF DATAFLOW ARCHITECTURES: FROM STATIC DATAFLOW TO P-RISC. *International Journal of High Speed Computing 05*, 02 (1993), 125–153.

[4] Arvind, and Culler, D. E. Annual Review of Computer Science vol. 1, 1986. Annual Reviews Inc., Palo Alto, CA, USA, 1986, ch. Dataflow Architectures, pp. 225–253.

[5] Arvind, and Iannucci, R. A. A Critique of Multiprocessing Von Neumann Style. In *Proceedings of the 10th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1983), ISCA '83, ACM, pp. 426–436.

[6] Arvind, and Nikhil, R. S. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers 39*, 3 (Mar 1990), 300–318.

[7] Arvind, Nikhil, R. S., and Pingali, K. K. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11*, 4 (Oct. 1989), 598–632.

[8] Arvind, A., and Gostelow, K. P. The U-interpreter. *Computer 15*, 2 (Feb 1982), 42–49.

[9] Ashcroft, E., and Wadge, B. Some Common Misconceptions About Lucid. *SIGPLAN Not. 15*, 10 (Oct. 1980), 15–26.

[10] Ashcroft, E. A. *Dataflow and Eduction: Data-driven and demand-driven distributed computation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, pp. 1–50.

[11] Ashcroft, E. A., Faustini, A. A., Jagannathan, R., and Wadge, W. W. *Multidimensional Programming*. Oxford University Press, Oxford, UK, 1995.

[12] Backus, J. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[13] BECK, J. P., PLAICE, J., and WADGE, W. W. Multidimensional infinite data in the language Lucid. *Mathematical Structures in Computer Science 25*, 7 (2015), 1546–1568.

[14] Benveniste, A., Caspi, P., Le Guernic, P., and Halbwachs, N. *Data-flow synchronous languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, pp. 1–45.

[15] Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U. A., and Pasquin, R. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 7:1–7:14.

[16] Bic, L. A Process-oriented Model for Efficient Execution of Dataflow Programs. *J. Parallel Distrib. Comput. 8*, 1 (Jan. 1990), 42–51.

[17] Bic, L., Roy, J. M. A., and Nagel, M. Exploiting iteration-level parallelism in dataflow programs. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems* (Jun 1992), pp. 376–381.

[18] Bifet, A., Holmes, G., Pfahringer, B., Kirkby, R., and Gavaldà, R. New Ensemble Methods for Evolving Data Streams. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2009), KDD '09, ACM, pp. 139–148.

[19] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 285–296.

[20] Buck, J., and Lee, E. A. The token flow model, 1992.

[21] Buehrer, R., and Ekanadham, K. Incorporating Data Flow Ideas into Von Neumann Processors for Parallel Execution. *IEEE Trans. Comput. 36*, 12 (Dec. 1987), 1515–1522.

[22] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1987), POPL '87, ACM, pp. 178–188.

[23] Chan, Y., Wellings, A., Gray, I., and Audsley, N. On the Locality of Java 8 Streams in Real-Time Big Data Applications. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems* (New York, NY, USA, 2014), JTRES '14, ACM, pp. 20:20–20:28.

[24] Charalambidis, A., Papaspyrou, N., and Rondogiannis, P. Tagged Dataflow: a Formal Model for Iterative Map-Reduce, 03 2014.

[25] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 21–21.

[26] Culler, A., Culler, D. E., and Ekanadham, K. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. In *Proceedings of the Conference on CONPAR 88* (New York, NY, USA, 1989), Cambridge University Press, pp. 541–555.

[27] Culler, D. E., and Arvind. Resource Requirements of Dataflow Programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1988), ISCA '88, IEEE Computer Society Press, pp. 141–150.

[28] Davis, A. L., and Keller, R. M. Data Flow Program Graphs. *Computer 15*, 2 (Feb. 1982), 26–41.

[29] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[30] Dean Brock, J. *Consistent semantics for a data flow language*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980, pp. 168–180.

[31] Dennis, J. B. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation* (London, UK, UK, 1974), Springer-Verlag, pp. 362–376.

[32] Dennis, J. B., and Misunas, D. P. A Preliminary Architecture for a Basic Data-flow Processor. *SIGARCH Comput. Archit. News 3*, 4 (Dec. 1974), 126–132.

[33] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 810–818.

[34] Ewen, S., Tzoumas, K., Kaufmann, M., and Markl, V. Spinning Fast Iterative Data Flows. *Proc. VLDB Endow. 5*, 11 (July 2012), 1268–1279.

[35] Faustini, A., and Jagannathan, R. Multidimensional Problem Solving in Lucid. Tech. Rep. SRI-CSL-93-03, Computer Science Laboratory, SRI International, 1993.

[36] Gajski, D. D., Padua, D. A., Kuck, D. J., and Kuhn, R. H. A Second Opinion on Data Flow Machines and Languages. *Computer 15*, 2 (Feb. 1982), 58–69.

[37] Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U. Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow. 2*, 2 (Aug. 2009), 1414–1425.

[38] Gelernter, D., and Carriero, N. Coordination Languages and Their Significance. *Commun. ACM 35*, 2 (Feb. 1992), 97–107.

[39] Ghosh, S., Bandyopadhyay, S., Mazumdar, C., and Bhattacharya, S. Handling of recursion in dataflow model. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge* (New York, NY, USA, 1984), ACM '84, ACM, pp. 189–196.

[40] Gupta, A., Mumick, I. S., and Subrahmanian, V. S. Maintaining Views Incrementally. *SIGMOD Rec. 22*, 2 (June 1993), 157–166.

[41] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE 79*, 9 (September 1991), 1305–1320.

[42] Iannucci, R. A. Toward a Dataflow/Von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Los Alamitos, CA, USA, 1988), ISCA '88, IEEE Computer Society Press, pp. 131–140.

[43] Indiana University, B. C. S. D., Friedman, D., and Wise, D. *CONS SHOULD NOT EVALUATE ITS ARGUMENTS.* Indiana University. Computer Science Department, 1975.

[44] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 59–72.

[45] Jaffe, J. M. The equivalence of R.E. program schemes and data flow schemes. *Journal of Computer and System Sciences 21*, 1 (1980), 92 – 109.

[46] Jagannathan, R. Adding Eagerness to Eduction. In *Seventh International Symposium on Lucid and Intentional Programming* (1994).

[47] Jagannathan, R. Dataflow Models. In *Parallel and Distributed Computing Handbook. McGraw-Hill* (1995).

[48] Jagannathan, R., Dodd, C., and Agi, I. GLU: A High-Level System for Granular Data-Parallel Programming. 63–83.

[49] Johnston, W. M., Hanna, J. R. P., and Millar, R. J. Advances in Dataflow Programming Languages. *ACM Comput. Surv. 36*, 1 (Mar. 2004), 1–34.

[50] Kahn, G. The semantics of a simple language for parallel programming. In *Information processing* (Stockholm, Sweden, Aug 1974), J. L. Rosenfeld, Ed., North Holland, Amsterdam, pp. 471–475.

[51] Lee, B., and Hurson, A. R. Issues in dataflow computing. *ADV. IN COMPUT 37* (1993), 285–333.

[52] Lee, B., and Hurson, A. R. Dataflow Architectures and Multithreading. *Computer 27*, 8 (Aug. 1994), 27–39.

[53] Lee, E. A., and Messerschmitt, D. G. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput. 36*, 1 (Jan. 1987), 24–35.

[54] Lee, E. A., and Messerschmitt, D. G. Synchronous Data Flow. *Proceedings of the IEEE 75*, 9 (Sept 1987), 1235–1245.

[55] Lee, E. A., and Parks, T. M. Readings in hardware/software co-design. Kluwer Academic Publishers, Norwell, MA, USA, 2002, ch. Dataflow Process Networks, pp. 59–85.

[56] Mcsherry, F., G Murray, D., Isaacs, R., and Isard, M. Differential dataflow.

[57] Mei, H. T., Gray, I., and Wellings, A. Integrating Java 8 Streams with The Real-Time Specification for Java. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems* (New York, NY, USA, 2015), JTRES '15, ACM, pp. 10:1–10:10.

[58] Mihaylov, S. R., Ives, Z. G., and Guha, S. REX: Recursive, Delta-based Data-centric Computation. *Proc. VLDB Endow. 5*, 11 (July 2012), 1280–1291.

[59] Murray, D. G., and Hand, S. Scripting the Cloud with Skywriting. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 12–12.

[60] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[61] Murray, D. G., McSherry, F., Isard, M., Isaacs, R., Barham, P., and Abadi, M. Incremental, Iterative Data Processing with Timely Dataflow. *Commun. ACM 59*, 10 (Sept. 2016), 75–83.

[62] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[63] Nikhil, R. S. Can Dataflow Subsume Von Neumann Computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*.

[64] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 1099–1110.

[65] Parks, T. M., and Roberts, D. Distributed process networks in Java. In *Proceedings International Parallel and Distributed Processing Symposium* (April 2003), pp. 8 pp.–.

[66] Pingali, K., and Arvind. Efficient Demand-driven Evaluation. Part 1. *ACM Trans. Program. Lang. Syst. 7*, 2 (Apr. 1985), 311–333.

[67] Pingali, K., and Ekanadham, K. Accumulators: New logic variable abstractions for functional languages. *Theoretical Computer Science 81*, 2 (1991), 201 – 221.

[68] Plaice, J. *Multidimensional Lucid: Design, Semantics and Implementation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 154–160.

[69] Rondogiannis, P., and Wadge, W. Intensional Programming Languages. In *First Panhellenic Conference on New Information Technologies* (Athens, Greece, 1998), pp. 85–94.

[70] Seo, J., Park, J., Shin, J., and Lam, M. S. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow. 6*, 14 (Sept. 2013), 1906–1917.

[71] Silc, J., Robic, B., and Ungerer, T. Asynchrony in Parallel Cmputing: From Dataflow to Multithreading. Tech. rep., 1997.

[72] Tesler, L. G., and Enea, H. J. A Language Design for Concurrent Processes. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 403–408.

[73] Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. Data-Driven and Demand-Driven Computer Architecture. *ACM Comput. Surv. 14*, 1 (Mar. 1982), 93–143.

[74] Ungerer, T., Silc, J., and Robic, B. Beyond Dataflow.

[75] Wadge, W. W., and Ashcroft, E. A. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[76] Watson, I., and Gurd, J. A prototype data flow computer with token labelling. In *Managing Requirements Knowledge, International Workshop on, vol. 00* (NEW YORK, June 1979), pp. 623–628.

[77] Zhang, Y., Gao, Q., Gao, L., and Wang, C. iMapReduce: A Distributed Computing Framework for Iterative Computation. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (May 2011), pp. 1112–1121.