A DATAFLOW LANGUAGE (AVON) AS AN ARCHITECTURE DESCRIPTION LANGUAGE (ADL)

Ashoke Deb
Department of Computer Science, Memorial University, Canada
ashoke@cs.mun.ca

Abstract: Avon is a dataflow graph language which insists single-assignment side-effect free paradigm. While it is an asynchronous system, the synchronicity is achieved by explicit events, thus allowing a sub-system to have local sychronization, while being globally asynchronous. A powerful facility in Avon is line filters where a predicate can be associated with input as well as output ports. These filters can screen values from the streams either at the source or at the sink.

We demonstrate that a small subset of Avon allows us to describe a computer architecture of substantial complexity in a natural and intuitive setting.

Keywords: dataflow, single-assignment, stream, filter, strictness, asynchronous, nondeterminism, pipelining, speculative execution, flushing.

1. DATAFLOW LANGUAGES [1, 2]

Tools shape our thoughts. The concept of dataflow is fundamentally different from that of control flow [3]. The discipline of thinking in terms of values about a solution of a problem may be a new experiences for a newcomer to this area.

As in other areas of computer science, designers of dataflow languages have different viewpoints – in terms of syntax, semantics and pragmatics of the language. These differences mostly originate from the philosophies or other considerations. For example, LUCID [4, 5] was originally designed as a declarative language which makes proving programs easy; Id [6] was designed to run on a fine-grain dataflow machine, called Tagged-token machine; Sisal [7] was designed as an applicative language to run on different multiprocessors systems; and Avon [8] was designed as a minimalist dataflow graph language.

1.1 Introduction to Avon [8]

Avon is a dataflow language with a small number of powerful constructs. Dataflow programs written using Avon are actually dataflow graphs [9]. We will introduce some of its main language constructs via simple examples.

In Avon, a line (or, a variable) associates with a stream of arbitrary length. Each output is singularly defined, and hence no side-effect is allowed.

Filters in Avon:

In Avon, a filter is a predicate which can be associated with a line – either an input line or an output line. The purpose of a filter is to 'remove' the current value from the line if the associated filter is not satisfied for the current values. A filter on an input line is a conditional expression which can involve the names of input lines, but not the names of output lines. A filter can also be attached to an output line, in which case the output line filter can involve the input line names and also the output line names. Absence of a filter with a line is the same as having a filter *TRUE,* which will allow values unfiltered.

Example 1: *To find the largest numbers so far from a stream of integers.*

An Avon program graph which computes the sequence of largest numbers found so far is shown at the top of Figure 1(a). An example of execution of the program is shown in four stages. The initial value in the line L is 0; and, say, the values appearing in line A are 5, 4, 3, 9, 7, 2 ... and so on. There is a *filter, (A > L),* attached to the line A. The purpose of this filter is to 'remove' the values in line A which do not 'pass through the filter' ie. do not satisfy the condition. In other words, if the 'current' value in A is less than or equal to the current value in L, then that value is removed from the line A, and the subsequent values in A will 'move forward'. For instance, the value 4 in line A gets removed, because $4 \not> 5$.

In *textual form* the Avon dataflow graph in Figure 1(a) will be written as:

```
LARGEST-SO-FAR:
        {IN A,L; OUT C; INIT L [0];
        INCOND A (A > L);
        A → C ;
        L ← C;}
```

Elements of Avon:

In Avon, **semicolon** (;) is used to imply relaxed ordering or *unordering.* This means that any two syntactic entities separated by a semicolon

(;) can be permuted without any effect on the program correctness. The *input* and *output* lines have distinct names. For example, *A* and *L* are used for two inputlines; and *C* is the output line. **IN**,**OUT**,**INIT** and **INCOND** are keywords. Input line names are given after the keyword **IN**. Output line names appear after **OUT**. An input line may have an *initial stream* of values, and these values are given between **square brackets** ([and]) following the keyword **INIT**. An *input predicate, or filter,* for an input line, appearsafter the keyword **INCOND**. Absence of a predicate (or, empty predicate) implies constant *TRUE.*

The *body* of the node contains one or more definitions, defining output(s) in terms of input(s). The symbol for definition is **right arrow** (→). An output name cannot have more than one definition.

A program may have a set of *line connectors* as well – where, a line connector connects an output line to an input line. The symbol used for connection is **left arrow** (←).

Types as filters and polymorphism:

In *Example 1(a)*, the input line A may carry any type of values, which is of course not what is specified. In Avon, types are treated as filters. There is a generalised predicative function **IS**, which can be used *to define type of a line as a filter.* In Figure 1(b) , the input line *A* has an additional filter *IS(A, INTEGER)* – the keyword **INTEGER** represents the set of all the integer values. This filter will accept only the integer values appearing on the line A.

One advantage of using types as line filters in dataflow is that we can associate types of a fairly complex nature, including *negative type,* to a name. For example, a filter *IS (A, COMPLEX OR REAL AND NOT INTEGER)* will allow all the reals and complex, but not the integers.

Using types as filters in this manner gives us a means of allowing values of different types to appear for an input or output line name (called *restricted polymorphism*). Of course, not having any type filter allows all types of values –ie. *unrestricted polymorphism.*

In *Example 1* the output from the line C will be a stream of values. One may wish to see only the ultimate largest number, but not largest numbers so far. In Figure 1(b), we do just that. **EOD** stands for end-of-data value. The output filter (A = EOD) will then absorb all the results appearing on the line G until the input line gets to the end-of-data value.

A note on the difference between Input Filters and Output Filters:

Although, at first glance, it may seem that with the availability of input filters the output filters are redundant; but they are not. The presence of a filter on an input line removes values from that line only if the associated filter is not satisfied, and before the node fires, thus keeping the values on the other input lines. Whereas, an output filter is

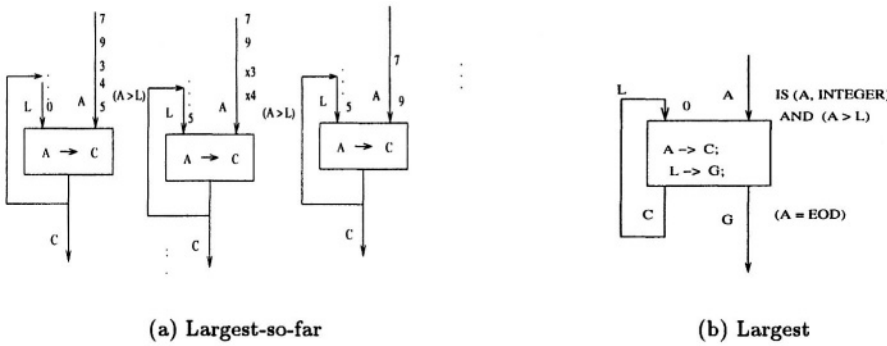

Figure 1. An example of Avon Program

"active" only after the node fires by which time the current values from all the input lines have been "consumed" (and essentially are removed from the input lines).

2. DESCRIBING MACHINE ARCHITECTURE

2.1 von Neumann machine and IPC

Since the introduction of APL (Iverson, 1962), there were about 200 different languages proposed, although only some of them are best known. For example, ISP (Bell and Newell, 1970), ISPL (Barbacci, 1976), ISPS (1977, Barbacci), SA* (Dasgupta, 1981), AADL (Damn, 1984), MIMOLA (Marwedei, 1984), VHDL (DOD, Intermetrics, IBM, TI, 1985) [10].

Most of these languages, with the exception of APL which is a functional and side-effect free language, are very much control flow oriented and imperative in their behaviour. Some of them succeed in classifying computers in terms of their processors, memory and switches. But most are specification languages to be used to automatically convert the specification into hardware, or to study performance. Some attempt to express the "dataflow" nature of a computer using control flow paradigm; some would express "parallelism" using cumbersome syntax, or introduce the concept of "stream" as an artifact. There are other systems proposed which deviate from traditional control flow style. For example, Johnson [11] suggested using recursive equations to describe digital design; Cardelli [12] suggested an algebraic approach to hardware description; and more recently Arvind and Shen [13] suggested using term rewriting system for description of processors.

Dataflow concepts have been applied in some languages, such as Kahn Network of Processes (KNP) [14], Synchronous Data Flow (SDF) [15] and SIGNAL [16]. These languages are useful in modelling signal processors. SDF do not allow asynchronicity, although KNP does. Also, unlike KNP, SDF insists on finite size streams. But, they do not restrict themselves to "pure" dataflow paradigm – eg. side-effects are allowed, and single-assignment is not necessary. SIGNAL differs from KNP in the sense that it allows "no value" ($\perp$) in the trace so that handling process deadlock (or, non-firing) can be accommodated in an unified manner. In SIGNAL, an elementary process produces an output value (at time t) by acting on all input values (at time t).

Avon, on the other hand, insists on single-assignment, absence of side-effect; it is asynchronous, and its streams are of arbitrary lengths. It does not insist on the output port to be empty in order for a node to fire. It allows independent "filters' on any of the input or output ports, which provides a powerful tool for modelling – eg, streams can be screened both by the destination (receiver) node as well as the source (sender) node. Synchronicity is achieved by explicit events, and thus different subsystems can synchronize locally, still maintaining global asynchronicity.

In this paper, we exploit only a subset of facilities available in Avon, and demonstrate that dataflow graph language provides a natural and transparent way of describing machines.

Traditionally, a von Neumann machine, executing one instruction per cycle, is described by a control-flow algorithm, known as Instruction Processing Cycle (IPC). For example, consider a machine with three instructions – *ADD, LW* and *BRZ.* Instructions are single address, with Accumulator (ACC) as the implied operand.

```
Repeat
    (* Sequential address generator stage *)
        PC = PC + 1;
    (* New Instruction generator stage *)
        IR = M[PC]
    (* Ins  truction Preparation and Issuing stage *)
        OP = DC (IR)
        OPR = IA (IR)
    (* Execution Stage *)
        Case OP of
            ADD: ACC = ACC + M[OPR]
            LW: ACC = M[OPR]
            BRZ: If (ACC = 0) Then PC = OPR
        Endcase
Forever.
```

In the example above, PC stands for the traditional program counter, M stands for Memory where both instructions as well as data reside, and for the time being can be viewed as an one-dimensional array; IR stands for the instruction register; DC and IA stand, respectively, for a functional unit which extracts the opcode (OP) and operand (OPR) from IR;

The IPC given above can be hierarchically expanded to introduce, for example, multilevel memory, complex instruction formats, addressing schemes and larger instruction repertoir.

It can also be used for simulation purposes, and datapath and control section designs.

But, the control-flow oriented description of IPC, and the related imperative semantics, make it very difficult to augment the IPC in a natural way to describe advanced concepts of overlapped instruction execution (instruction pipelining), data pipelining, multifunctional machines with multiple instruction issuing, pipeline bubbles, branch hazards and "flushing" pipeline stages, conditional issuing of instructions etc.

2.2 Using Avon as Architecture Description Language

In this paper, we demonstrate, as an example, how Avon can be used to describe a pipelined machine with speculative execution. We will start with a machine which is *not pipelined,* has only ADD and LW type instructions, and does not have branch instructions. Then we will show that this description can easily be augmented to turn that machine into a pipelined machine. Following that we will include a branch type instruction, BRZ, to the pipelined machine, and show how branch hazards are represented.

Non-pipelined Sequential machine, with no branch type instructions. Figure 2(a) describes a strictly sequential von Neumann machine.

Note that this description is very similar to the imperative style IPC given earlier, although with several significant differences.

Avon, being single assignment language, cannot have more than one function assigning values to the same output line, nor can it have the same line name both as input as well as output. Therefore, the imperative construct like PC = PC + 1 will be illegal. Similarly, both of ACC = M[OPR] + ACC and ACC = M[OPR] are not allowed.

Two lines are shown joined, explicitely, by a non-deterministic OR, or, sometimes implicitely, by simple fan-in.

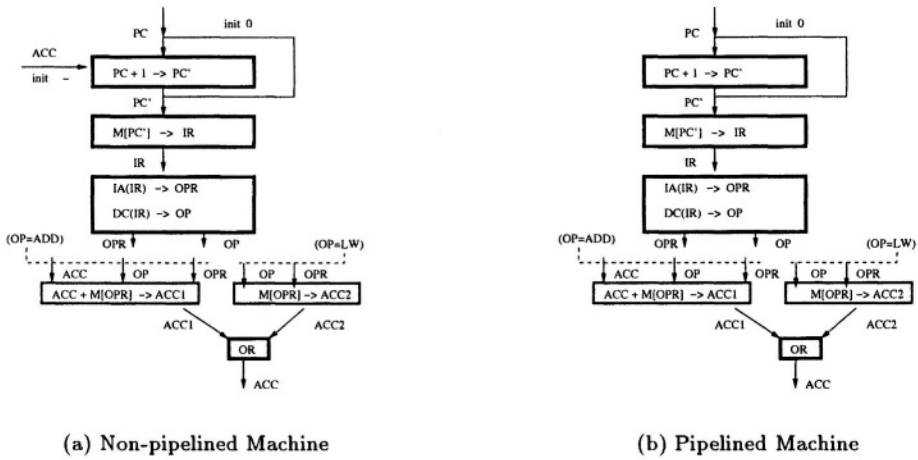

Figure 2. Non-pipelined and Pipelined Machine - no branch instructions

To further save us from redundant drawings, when two lines are labeled with the same name, it means that they represent the same physical line.

The input condition attached to all the input lines, OP, OPR and ACC of the node which computes ADD, is (OP = ADD), meaning that if the current value of OP is ADD, then this node will "fire" and thus will produce a value via ACC1; if the condition is false, then the current values from these lines of this node will be "absorbed" or discarded. Similarly, for the node which computes LW.

In order to "sync" the address generation station so that it generates a new value only when the previous instruction is complete, we have an "extra" input line ACC to the node $PC + 1 \rightarrow PC'$, which although has no effect on the value generated, has the effect on the node's firing intervals. Note that the "sync" line did not have to be fed from ACC, it was just convenient in this example.– it simply mimics the behaviour of "repeat forever" construction of the imperative style IPC shown earlier.

Pipelined machine with no branch type instructions. A simple modification of the first diagram (describing a sequential machine) gives the description of a pipelined machine – we had to remove the "sync" input line ACC from the address generation stage!! (See Figure 2b).

With this modification, the address generation stage produces a stream of addresses which get buffered in the input line to next stage

- the instruction generation stage. So, the input line PC' refers to the address cache. Using these addresses from the line PC', the instruction generation stage produces a stream of instructions which get buffered in the input line IR of the instruction preparation stage, which is usually referred to as instruction cache. And so on.

Note that now it is not difficult to see how the instructions are "moving along" the stages of the pipeline, which was a difficult proposition in imperative style description.

These interstage buffers can be either multilevel storage structures (to reflect its infiniteness), or its finite implementation can be done by proper annotation and demand/acknowledge signals.

Note that the presence of two distinct nodes in the execution stage (one for computing ADD instructions and the other for computing LW instructions) clearly reveals the possibilities of issuing multiple instructions to a multifunctional machine, Of course, in that case the data dependencies among instructions will have to be taken into consideration.

If we are to restrict ourselves to pipelined but single issue machines, then that can be easily described by introducing a "sync" line ACC to the instruction issue stage.

Pipelined machine with branch type instructions . In addition to the system described in the last section, let us introduce a branch type instruction BRZ (defined earlier). To keep the matter simple, we will make a fair and realistic assumption that each stage takes equal amount of time, and hence each stage may have at most one value.

In a pipelined machine, if a branch instruction is successfully executed, a number of things will have to considered – the target address will be the address of the next instruction to be executed; and as a result of which, all previously generated addresses and the instructions which are already in the pipeline have to be discarded. (Please refer to Figure 3.)

Modifying the execution stage: With the introduction of this new instruction, we modify the execution stage by introducing two nodes (see parts 1 and 2.)

The first node computes if the branch condition is true. And then, if the branch is true (indicated by a T token on the *branch* line, the following node computes OPR → PC" In fact, if the branch is evaluated to be true, a four value stream (T, F, F, F) will be generated. Reader can ignore this for the time being without loss of continuity, and it will be explained shortly.

Updating the next PC value to be the value of PC": Now, PC' and PC" are OR-ed (multiplexed) using the value of the line *branch.*

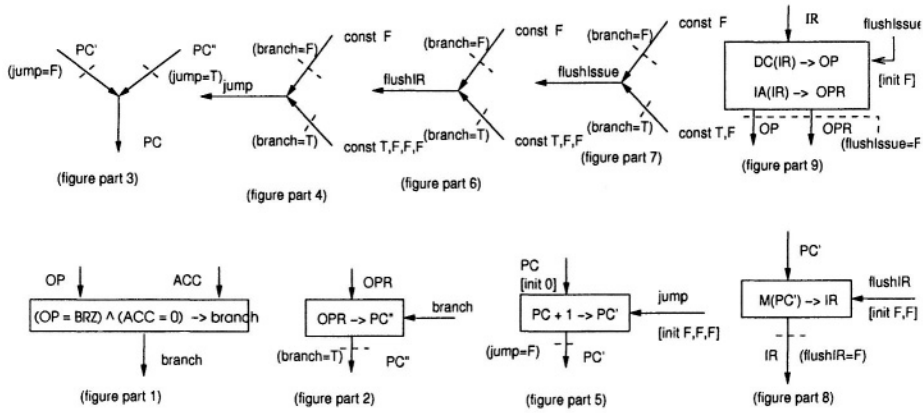

Figure 3. Segments of the Avon description of a speculative pipelined machine with branch instruction

Initially, the value on the line *branch* would of course be F (false), and as long as the current instruction executed is a non-branch type, then a stream of F values will be on *branch* line, allowing sequential addresses to be generated. If the value of *branch* becomes T, then the value on PC" comes to PC. (See parts 3, 4 and 5.)

Flushing all intermediate stages of the pipeline:

In order to "flush" all the intermediate stages, ie "discard" all the values from the internal lines PC' (gateway to instruction generator), IR (gateway to instruction issuing stage) and the lines OP and OPR (gateway to the execution stage) we associate a new line condition *(branch = F).* Thus, if and when *branch* value becomes true, the values from PC', IR, OP and OPR will be discarded. (See parts 6 and 7.)

As a consequence though, this will create "bubbles" in the three stages, and so no further values for *branch* will be generated. To make these bubbles progress through the stages, we need extra F values following a T value on the *branch* line.

Similarly, to initially "fill" the pipe, the *init* values of *branch* are to be set accordingly for each stage of the pipeline, and are shown on the diagram.

Finally, putting together the entire description is simply a matter of connecting the appropriate lines. (Please note that the ADD and the LW units were left out from the diagram for space.)

3. CONCLUSION

In this paper, we have attempted to show that Avon, a dataflow graph language, is not only a powerful language for general purpose computa-

tion, but it also proved to be useful in describing machine architecture in a natural and intuitive way that is not possible with control flow based imperative languages.

Our future research will focus on describing other aspects of machine architecture and constructs of communicating sequential processes.

REFERENCES

[1] W. B. Ackerman, Dataflow Languages, *IEEE Computer,* Feb, 1982.

[2] A. Deb, Data Flow Languages, In *Encyclopedia of Library and Information Science,* Vol. 66, Marcel Dekker, 2000.

[3] M. Broy, Ed., Control Flow and Data Flow - concepts of Distributed Programming, Springer-Verlag, vol. 14, 1985.

[4] E. A. Ashcroft et. al., Lucid - A formal system for writing and proving programs, *SIAM J. Comp.,* 5, pp. 519 - 526.

[5] E.A. Ashcroft and W. W. Wadge, Lucid, the Dataflow Programming Language, Academic Press, 1985.

[6] R. S. Nikhil, The Parallel Programming Language Id and its Compilation for Parallel Machines, In *Proc. of the Workshop on Massive Parallelism: Hardware, Programming and Applications,* Academic Press, 1990.

[7] J. R. McGraw, et. al., SISAL: Streams and Iteration in a Single Assignment Language, Reference Manual 1.2, M-146, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[8] A. Deb, Avon: A Dataflow Language, In *Second International Conference on Supercomputing,* Florida, USA, pp. 9 -19, International Supercomputing Institute, 1987.

[9] A. L. Davis and R. M. Keller, Data Flow Program Graphs, *IEEE Computer,* pp. 26 - 41, Feb. 1982.

[10] Lipsch R. et al, VHDL: Hardware Description & Design, Kluwer, 1989.

[11] S. Johnson, Synthesis of Digital Design from Recursive Equations, MIT Press, 1983.

[12] L. Cardelli, An Algebraic Approach to Hardware Description and Verification, Ph.D dissertation, Univ. of Edinburgh, 1982.

[13] Arvind end Shen, Using Term Rewriting Systems to Design and Verify Processors, *IEEE Micro,* pp. 36-46, June 1999.

[14] Kahn, G. The semantics of a simple language for parallel programming, In *Information Processing 74,* pp. 471-475, North-Holland, 1974.

[15] Lee, E. A. et all, Synchronous Data Flow, In *Proc. of IEEE,* pp. 55-64, Sept 1987.

[16] Gautier, T. et al, SIGNAL: A declarative language for synchronous programming of real-time systems, In *Conference on Functional Programming Languages and Computer Architecture,* pp.257-277, LNCS, 274, Springer-Verlag, 1987.