Towards a Generic Dataflow Architecture

R. Jagannathan
SRI International
Menlo Park, California 94025
and
E.A. Ashcroft
Arizona State University
Tempe, Arizona 85287

Consider the following question:

Are data-driven and demand-driven execution different computing strategies \(^1\) or are they mechanisms that implement computing strategies?

We espouse the second view, i.e., data-driven and demand-driven execution are mechanisms that implement computing strategies. This view, as it turns out, allows us to develop a generic dataflow architecture — one capable of embodying any dataflow computing strategy.

First, we assert that any dataflow computing strategy can be described in an abstract framework without appealing to the data-driven and demand-driven execution mechanisms. Second, we assert that whether an operation is data-driven or demand-driven need only depend on a property of the operation and not on the underlying computing strategy, and that demand-driven is a more general execution mechanism than data-driven. These two assertions allow us to describe the basis for a generic dataflow architecture that is basically demand-driven with data-driven execution being used when appropriate. Such an architecture can embody diverse computing strategies with equal ease.

\(^1\)We use the phrase "computing strategy" synonymously with the phrase "computing model."

1
1 Abstract Framework

We will assume that programs are expressed in the language of flat operator nets: a syntactically restricted form of operator nets [4]. (The proposed framework extends to the language of general operator nets although we do not consider it here.) In the flat operator nets language, a program is a directed graph each of whose vertices is either an input vertex, an output vertex, an operation vertex, or a split vertex. An input vertex has no incoming edge and one outgoing edge; it allows interaction with the input world. An output vertex has one incoming edge and no outgoing edge; it allows interaction with the output world. An operation vertex has several incoming edges and exactly one outgoing edge; it denotes an operation that is applied when the vertex is executed. A split vertex has exactly one incoming edge and several outgoing edges; it allows for reuse of a computed result.

It is useful to think of an edge as representing a stream of datons where each daton is a place-holder for a value. A daton can be uniquely identified by the edge it belongs to and its position relative to other datons. A vertex can be thought of as a producer and/or a consumer of daton values. The input vertex produces values for datons from the input world whereas the output vertex consumes values of datons for the output world. The operation vertex consumes the necessary daton values on its incoming edges and produces a resultant daton value on its outgoing edge. The split vertex retains daton values on its incoming edge until they are provided on each of the outgoing edges.

In operational terms, a program can be viewed as a mapping of streams of values of input datons to an stream of values of output datons. We refer to the execution of a program with a specific stream of input daton values and specific needs for output daton values as a computation. We can think of computation as proceeding in stages. At the initial stage (stage 0), all datons of the program are undefined (i.e., they do not have a value) except possibly some of the input datons. An undefined daton is definable at a given stage if all the datons that it directly depends on are defined at the previous stage. The underlying computing strategy determines which of the undefined datons that are definable at a given stage, will be defined at that stage. Therefore, at each subsequent stage of the computation, some of previously undefined datons become defined. We claim that different
computing strategies differ only in the following two respects.

1. which of the datons of a computation will ever be defined and

2. when (i.e., at which stage) will such datons be defined.

We introduce the abstract notion of "daton desire" where a definable daton can be defined only after it has been desired. Furthermore, a daton once desired remains desired regardless of whether it is defined or not. This allows us to describe a computing strategy in precise and abstract terms. In particular, a computing strategy can be uniquely characterized by the following two choices it makes.

1. which datons of a computation are desired and

2. for each such daton, at which stage is it desired.

We briefly show how computing strategies embodied by extant dataflow machines can be described using our framework. It turns out that a dataflow machine either computes as many daton values as possible as soon as they are definable or it computes those daton values that it knows are useful. We can classify the underlying computing strategy as either eager or lazy. An eager computing strategy attempts to define as many datons of a computation as possible as soon as possible whereas a lazy computing strategy only defines those datons of a computation that are useful in defining needed output datons.

Orthogonal to this, some dataflow machines compute daton values by increasing order of their conceptual positions while others compute daton values in any order. We say that the underlying computing strategy is either be piped or tagged. A piped computing strategy imposes that a daton can be defined only after its predecessor is defined whereas a tagged computing strategy imposes no such ordering.

We describe three commonly-used computing strategies, namely, piped eager, tagged eager, and tagged lazy using the abstract framework. (A fourth computing strategy is piped lazy which we do not consider here.)

**Piped Eager**

- At the initial stage, desire initial datons of all edges.
• At a given stage, desire a daton if and only if its predecessor is defined.

The piped eager computing strategy is embodied by the implementations described in [6,9,5].

Tagged Eager

• At the initial stage, desire all datons on all edges.

The tagged eager computing strategy is embodied by the implementations described in [1,7,8].

Tagged Lazy

• At a given stage, desire an output daton if and only if the external world needs it at that stage.

• At a given stage, for each desired daton, desire at the next stage all datons that it is known to directly depend on.

The tagged lazy computing strategy is embodied by the implementation described in [3].

We see that it is possible to describe dataflow-based computing strategies using an abstract framework without appealing to data-driven or demand-driven execution mechanisms. We believe that this is significant because it not only allows various properties of computing strategies to be uniformly compared, but it also removes implementation considerations from computing strategies themselves.

2 Execution Mechanisms

Given that a dataflow-based computing strategy can be described in an abstract framework, we now consider how such a strategy can be implemented using data-driven and demand-driven execution. We assert that the computing strategy itself has no bearing on which execution mechanism to use. Whether to use data-driven or demand-driven execution is need not be a global choice but a choice that can vary with each vertex. Specifically, the
choice is determined by whether the vertex denotes an operation that is *predictable* or not.

We say that an operation is predictable if for any particular resultant daton value, it is possible to predetermine (relative to the position of the daton) all argument daton values that are necessary to produce it. Otherwise, the operation is considered unpredictable. For example, sequence arithmetic and logical operations are predictable; the operation *fy* is predictable as is the operation *next*. An example of an unpredictable operator is *if-then-else* because for any resultant daton value, it is not possible to predetermine which one of the *then* or *else* argument-daton's value is necessary; that choice depends on the test argument-daton's value. Other examples are *whenever* and *upon*.

Essentially, predictability of an operation is the ability to predict whether a particular resultant daton value is definable based on the availability of argument daton values. With data-driven execution, an operation is applied and the resultant daton value defined when the necessary daton values are available. Therefore, data-driven execution is natural for predictable operations.

An unpredictable operation has the property that either it is not self-evident which daton value is being defined or some of the argument daton values need to be examined in order to determine what is the necessary set of datons whose values are required to define a resultant daton value. Demand-driven execution naturally provides for this: the resultant daton value to be defined is known from the demand itself and possibly some argument daton values are demanded and examined before demanding other argument daton values.

We see that whether to use data-driven or demand-driven execution is determined at the operation-level by a particular property that we call *predictability*. Demand-driven execution can be used effectively with both predictable and unpredictable operators whereas data-driven execution can be used effectively with only predictable operators. Hence, demand-driven execution is more general than its data-driven counterpart.
3 Basis for a Generic Architecture

Data-driven execution is useful from an efficiency standpoint – it does not require propagation of demands. Demand-driven execution is useful from a generality standpoint – some operations cannot be data-driven effectively. We develop the basis for a generic architecture using data-driven and demand-driven execution. The architecture is generic because it can embody any computing strategy that can be described in the abstract framework.

The eager computing strategy has been only implemented using data-driven execution only [2,10]. (Note that unpredictable operations such as if-then-else are replaced by “operationally-equivalent” fragments whose semantics are fuzzy.) It is useful to show the correspondence between the implementation and the underlying computing strategy. When an operation is data-driven, it is implicit that the resultant daton value is “desirable.” As a result, any daton value that is definable is also desirable. Therefore, if all operations are data-driven, all daton values which can be defined are implicitly desirable. This captures the essence of the eager computing strategy.

Similarly, the lazy computing strategy has only been implemented using demand-driven execution [3]. When an operation is demand-driven, the desire for the resultant daton value is made explicit. The daton desirability rules of the lazy computing model manifest themselves as demand-propagation in the implementation.

We claim that it is possible to use a combination of demand-driven and data-driven execution to implement either computing strategy, and in general, any computing strategy. The essential idea is that the overall implementation uses demand-driven execution with data-driven execution used where appropriate.

A desire for a daton is manifested as an explicit demand if the vertex that is to produce the daton is an input vertex, a split vertex, or an unpredictable operation vertex. Otherwise, the desire for the daton is implicitly demanded. Consequently, vertices whose outgoing datons are explicitly demanded are demand-driven whereas vertices whose outgoing datons are implicitly demanded are data-driven. (The corresponding edges are referred to as demand-driveable and data-driveable.)

An implementation of a computing strategy identifies which daton val-
ues of a computation need to be demanded and when these values need to be demanded. The demand for a daton value is made explicit if the associated edge is demand-driveable; otherwise, the demand for the daton value is implicit.

The difference between implementations of tagged eager and tagged lazy computing strategies is not that one is data-driven and the other demand-driven. With tagged eager, all daton values are demanded (as per the strategy) initially whereas with tagged lazy, daton values are demanded when they are known to be useful. In both implementations, a demand for a daton value is explicit only if the associated edge is demand-driveable; otherwise, the demand is implicit. Therefore, both implementations are based on a combination of demand-driven and data-driven execution although they embody totally different computing strategies.

4 Conclusions

We have argued that data-driven and demand-driven execution mechanisms should not be thought of as bases for dataflow-based computing strategies; rather, they are mechanisms used to implement dataflow-based computing strategies. To this end, we have shown how computing strategies can be described in an abstract, uniform framework without appealing to these mechanisms. We have also shown how whether to use data-driven or demand-driven execution mechanism need only be a property of the operation. We have developed the basis for a generic architecture that uses data-driven and demand-driven execution in which any computing strategy described in the framework can be effectively incorporated.

References


