Δ-DATAFLOW NETWORKS FOR EVENT STREAM PROCESSING

Rajit Manohar
Computer Systems Lab
Cornell University
Ithaca, NY 14853
Email: rajit@csl.cornell.edu

K. Mani Chandy
Computer Science 256-80
California Institute of Technology
Pasadena, CA 91125
Email: mani@cs.caltech.edu

ABSTRACT
We present the design of a new class of dataflow-like networks suitable for detecting complex conditions in systems where parameters change rapidly. Such networks are helpful for detecting conditions that signal threats or opportunities in areas such as logistics, finance, and public health. Examples of such applications are detection of money laundering, epidemics, and unauthorized intrusion into systems. We call these networks Δ-dataflow networks because nodes in the network propagate only changes in data values. We show how ultra low power asynchronous architectures that have been developed for sensor networks can provide an extremely efficient platform for executing such networks.

KEY WORDS
sensor networks; dataflow computation; event processors; asynchronous design

1 Introduction

Our world is increasingly instrumented with sensors, Radio-frequency IDs (RFIDs); smart homes monitoring temperature, humidity, movement and appliances; vibration sensors monitoring earthquakes; stock and commodity markets; and news services generate streams of events. The number of event generators continues to increase, and the potential pace of event generation per sensor is increasing.

Event processors correlate multiple event streams to generate events that signal complex conditions [4, 15]. The output from event processors may feed other event processors. Actuators respond to conditions by changing parameters that influence the environment. For example, an actuator may increase the thermostat setting in an air-conditioned office when the price of electricity increases above a threshold. Or an actuator may buy a certain amount of stock when its price dips below its long-term moving-point average. Sensors generate event streams, event processors consume event streams and generate value-added event streams, and actuators consume event streams and execute appropriate actions. Sense-and-respond applications are compositions of sensors, event processors and actuators [4]. An increasing number of consumer and enterprise applications are sense-and-respond applications. We propose an abstraction called Δ-*dataflow networks* for specifying and designing event processors. This abstraction is suitable for designing event processors, and in particular event processors that can be implemented efficiently using asynchronous VLSI techniques.

Some sensors may send messages periodically whether the parameters being monitored change or not. For the purpose of this paper, we restrict attention to messages that signal change in value. Thus we make the assumption that event streams are *stuttering-free*—i.e., no two consecutive values in the event stream are identical. Under this model, a stock market ticker source would only produce an output (it's current stock value *without* the time component) if the stock value changed. This implicit notion of only propagating changes builds in the notion of filtering redundant information into the computation model. As a consequence, it becomes important that this event information is communicated in a timely manner, since consumers of information do not know if a new value is produced until the information reaches them. This differs from standard soft-state based announce-listen protocols [5, 6], where periodic updates allow a consumer to know that the information is current.

Computation is performed on these event streams to generate new streams. These streams travel over communication links to other computation blocks until they reach their final destination. The collection of sources, destinations, computation nodes, and communication links can be thought of as a graph. These graphs most closely resemble dataflow networks [10].

While these networks can be implemented using conventional methods, the most efficient implementation of such networks is realized when the underlying implementation is also event-driven all the way to the hardware implementation. Asynchronous (or clockless) VLSI systems completely data-driven, and are therefore extremely well-suited for event-driven operation. Instead of using a periodic synchronization signal (the clock) to sequence steps in the computation, asynchronous systems use local handshaking for synchronization and sequencing [17].

Efficient asynchronous systems can operate with extremely low power budgets, and are very well suited for sensor network applications. We have developed an ultra low power processor that is optimized for sensor network operation that requires on the order of tens of picoJoules of energy per instruction [8, 11]. The combination of Δ-dataflow networks and ultra low power asynchronous architectures provides an extremely attractive methodology for the design of algorithms for event detection and data fusion in sensor network scenarios. We show how Δ-dataflow networks can be implemented on an asynchronous architecture.

In this paper we present a model that can be used to specify networks of event processors, and illustrate their operation with some examples (Section 2). A few key properties of these networks are described (Section 3), that allow us to reason about their aggregate behavior in common circumstances. We describe how such architectures can be implemented extremely efficiently using an asynchronous VLSI execution engine (Section 4), resulting in a system that is extremely power efficient.

Relation to Dataflow Networks. Dataflow networks are described by a graph in which vertices correspond to computation nodes, and edges correspond to communication channels that may or may not have buffering capability. Values are thought of as *tokens* that travel through the network along the channels. Nodes in the computation operate on tokens arriving on their inputs to produce new tokens on their outputs. Sophisticated behaviors can be described by combining nodes of different types, and there is a wealth of research on models, languages and compilers, and machine architectures for dataflow computation [1, 7]. In fact, the execution core of a modern out-of-order microprocessor (e.g., [19]) resembles the core of a dataflow architecture.

Dataflow networks were first introduced as a model of parallel computation in the early 1970's [10]. The original evaluation model is referred to as a "Kahn-McQueen" network. Since then, many variants of these networks have been investigated. Dataflow networks can be data-driven or demand-driven. In a data-driven network, data values travel from producers to consumers, and the arrival of data triggers computation that produces a new output. In demand-driven networks, requests for results travel from consumers back to producers, and these requests trigger the generation of outputs once the appropriate input requests are satisfied. Dataflow networks can also be either static or dynamic. In a static network,communication links are first-in first-out (FIFO) lossless channels. The output result is produced when a token arrives on each input, or in some special control-flow nodes, when tokens arrive on some (deterministic) subset of the inputs. In dynamic dataflow networks, the FIFO channel constraint is relaxed. Instead, each token contains a tag that identifies the specific operation it is associated with. A tagged output is produced when tokens are available on the appropriate set of inputs and have matching tags. For instance, under this classification, Kahn-McQueen networks are static, demand-driven dataflow networks. The networks investigated by Dennis are static, data-driven networks, while those investigated by Arvind are dynamic, data-driven networks [1, 7]. Highly pipelined asynchronous circuits can be thought of as direct implementations of static data-driven dataflow networks. In general, asynchronous circuits can also directly implement static demand-driven dataflow networks.

The difference between Δ-dataflow and other forms of dataflow computation is twofold. First, in a Δ-dataflow graph, an output token is produced only when the value computed differs from the previous output. Therefore, a new input token will only arrive at a particular node when the value on that input differs from the previous token that was received on that particular input. To examine some consequences of this constraint, consider a simple node that has two inputs A and B and produces the sum of those two inputs on its output C. This behaves as expected in a static dataflow model: whenever a token arrives on both inputs A and B, the tokens are consumed and the sum of the received inputs is produced as an output token on C. If the two tokens are 1 and 2 respectively, the output token will be 3. Now, what if the next set of inputs is 1 and 4? In a static dataflow model, the result 5 would be produced as expected. However, in a Δ-dataflow network, the 1 token would *never arrive on input A* due to the absence of stuttering! To resolve this, we come to the second difference between Δ-dataflow and traditional dataflow networks. A Δ-dataflow node is highly non-deterministic, and it can process a token on *any* input without waiting for additional tokens on any other input. The non-determinism arises because computation nodes do not know apriori if they will receive a token on a specific input, as tokens are discarded in a data-dependent manner. This difference will become obvious in Section 2, where we provide a specification of Δ-dataflow nodes.

Relation to Rete Networks. A Rete network is an efficient way to execute the match operation required in production-based logical reasoning systems [9]. A production system consists of a set of rules, each containing a premise and a set of actions. The system operates by using its knowledge base to determine those premises that are satisfied, and then invoking the actions of some chosen rule using a decision procedure to pick at most one rule. The Rete algorithm constructs an acyclic network using the premises of the rules, where nodes in the network correspond to premises (or parts of premises), and the leaves correspond to actions. The knowledge base is passed through the network, and if any part of it reaches a leaf then that leaf node action is enabled. This approach eliminates duplicate match operations since common predicates in multiple rules are only computed once. Also, new knowledge does not require recomputation of the entire match operation. Instead, only a small part of the network is activated, leading to an efficient incremental update of the set of enabled actions.

Rete networks resemble Δ-dataflow networks because of the idea of incremental updating. Partial results are stored in the nodes in Rete networks so that only changes are propagated. The difference is that Δ-dataflow networks are more general because we allow arbitrary functions in the network as well as local state information. A Rete network node is limited to operations such as unification and extracting predicates from the knowledge base (corresponding to "source nodes" below).

Δ-dataflow networks can be thought of as providing a structured form of memoization. Memoization can be found in a wide variety of algorithms and computation structures. Rete networks are but one instance of this concept. Dynamic programming is a general technique that uses memoization to avoid recomputing partial results, and can lead to significantly improved time complexity in exchange for increasing the space requirements of an algorithm. Any algorithm represented using dynamic programming (for a fixed problem size) can be easily computed using its dataflow graph, assuming that the relation between problems and sub-problems can be statically determined [13]. To compute the result when the inputs to the problem are changed, one can simply present the new input values to the dataflow network and the result will be recomputed by sharing common sub-computations. What makes Δ-dataflow networks more interesting is that *updates* to the problem automatically propagate to the result as well. Only those sub-problems that are data-dependent on the changed value are affected, and the rest of the computation is not repeated.

2 Δ-dataflow Model

A Δ-dataflow network is specified as a finite graph. Each vertex is a node that performs some computation. The edges correspond to FIFO channels with zero or finite buffer space. channels. Based on the graph, we can classify a vertex into exactly one of three categories:

- those with in-degree zero. These are *source* nodes, and correspond to sensors that intermittently generate new information.
- those with out-degree zero. These are *sink* nodes, and they correspond to the result of the computation.
- Those with non-zero in-degree and out-degree; these form the compute nodes.

Every Δ-dataflow network compute node is specified by its local state s and two functions: the state-transition function g, and the output function f. The compute node also has a copy of the last value received on every incoming edge in a vector $\mathbf{x}$=(x_1,. . .,x_n) (where the compute node has n input edges). The compute node also saves away y, the last value it produced on its output. For simplicity, we assume that when a compute node produces a value, the value is copied to all outputs.

When a node receives a new value on its ith input, the following operations are performed. (i) The value x_i is updated with the newly received input value; (ii) The node makes an internal state transition by replacing s with $g(s, \mathbf{x})$; (iii) The value $f(s, \mathbf{x})$ is computed as the new output. If this value differs from y, then the new value is propagated along the output edges to other compute/output nodes, and y is updated with the new value of the output. To prevent multiple output values in response to a change in input, we impose a fixed point requirement on function g, namely that

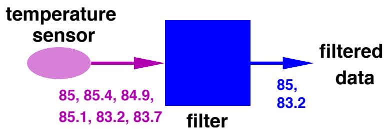

Figure 1. Threshold filtering of sensor data.

$$g(s, \mathbf{x}) = g(g(s, \mathbf{x}), \mathbf{x}) \qquad (1)$$

The fixed point requirement ensures that the assignment $s := g(s, \mathbf{x})$ used to compute the new state cannot be repeated to change s.

Assuming that the initial values for s, $\mathbf{x}$, and y are appropriately selected, the following properties hold once steps (i)–(iii) are completed for a compute node:

- $s = g(s, \mathbf{x})$ (from the fixed point requirement);
- $y = f(s, \mathbf{x})$;
- $\mathbf{x}$ is a vector of the last values received on the input ports;
- y is the last value produced on the output (modulo initialization).

Initialization. We permit any initialization of the system as long as the properties listed above hold, and the value of y associated with the output of a compute node matches the corresponding x_i value at the appropriate input (if any).

Example 1. Consider a sensor that produces a temperature reading, and we wish to filter this information so that we only obtain a reading when the temperature changes by one degree. To specify this, we use a compute node with one input and one output. The local state s is the last output y, and $g(s, x) = f(s, x)$. The function f is given by:

$$f(s, x) = \begin{cases} s & \text{if } |s - x| < 1 \\ x & \text{if } |s - x| \geq 1 \end{cases}$$

This functionality is shown in Figure 1.

Example 2. If we wish to calculate the average temperature across two sensors, a compute node that can perform this task does not require any local state s or g function (for instance, we can assume $s = 0$ and $g(s, x_1, x_2) = 0$), and the function $f(s, x_1, x_2) = \frac{1}{2}(x_1 + x_2)$. Figure 2 depicts such a system. Note that while the sequence of temperature

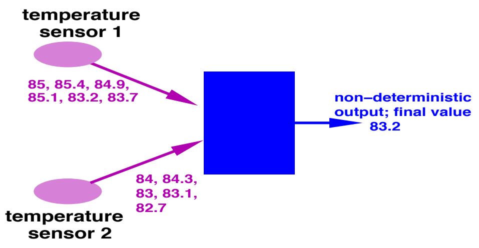

Figure 2. Averaging sensor data.

values produced on the final output depends on the order in which the updates are received from the sensors, the final value is the same.

To reduce the volume of output, we can use the filter from the first example to only provide new temperature readings for large changes in temperature. The result is shown in Figure 3. This illustrates the compositionality of the model, where we can use compute nodes together to start constructing complex change-detection conditions or aggregation operations.

3 Model Properties

The basic property of each compute node in Δ-dataflow networks is the implicit filtering operation on the output. This property can be captured by the following lemma.

Lemma 1 *For a compute node, the number of output updates is at most the sum of the number of input updates.*

Proof: Initially the number of input and output updates is zero. The sequence of operations performed in response to the receipt of one new input value can produce at most one new output value, concluding the proof. ∎

For applications such as data fusion or filtering, we expect that the Δ-dataflow network corresponds to an acyclic graph, and we are normally interested in the final values of all the outputs when no more changes need to be propagated along the edges in the network. We say that the network is quiescent when there are no changes to be propagated. Under such circumstances, we can easily establish the following result using Lemma 1.

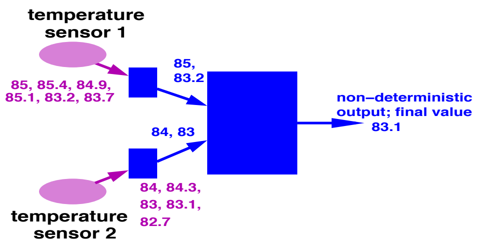

Figure 3. Averaging filtered sensor data.

Theorem 1 *If all source nodes produce a finite sequence of tokens, an acyclic Δ-dataflow network will eventually reach a quiescent state.*

Proof: Assume, toward a contradiction, that the network never reaches a quiescent state. An infinite sequence of node computations necessarily requires an infinite number of outputs to be propagated along the edges of the network. However, using Lemma 1, we can show that this is impossible. By assumption, the number of updates propagated from any source node is bounded. By Lemma 1, every node that receives inputs from sources only (these nodes exist because otherwise the graph is not acyclic) can only produce a finite number of output updates. Repeating this argument concludes the proof. ∎

In Example 2, we used an averaging operation in the computation graph. The compute node did not use any state information to produce its output. If the state s is not used to compute $f(s, \mathbf{x})$ in every Δ-dataflow node, then the computation is said to be stateless and we use $f(\mathbf{x})$ to represent the output computation.

Theorem 2 *A quiescent, stateless, acyclic Δ-dataflow network has a unique final state.*

Proof: Every source node has a unique final output value, since sources are deterministic. Erase all source nodes, and examine the rest of the graph. There will be a new set of nodes with in-degree 0 since the graph is acyclic. These nodes all have a deterministic $\mathbf{x}$, from the properties of a compute node given in Section 2. Therefore, by Section 2, all these nodes have a deterministic $y = f(\mathbf{x})$ as they are stateless, so the last value sent on the output is deterministic for these nodes as well. Repeating this argument concludes the proof. ∎

4 Implementation Strategy

The operations described above for sources, sinks, and nodes can be easily implemented using a sequential algorithm on a conventional architecture. In this section we discuss an implementation on a low power highly parallel asynchronous architecture that we have developed for parallel simulation.

Architecture Overview. We have created the "network on a chip" (NoC) [11], an asynchronous chip multiprocessor. The architecture was originally designed to support high-performance parallel discrete-event simulation [12]. Each processor has its own private memory, and communicates with the other processors only by passing messages via a highly-pipelined interconnect. We estimate that a single NoC chip will contain approximately one hundred processors, providing a highly parallel platform for the implementation of Δ-dataflow networks. A sixteen processor version of the NoC (4×4 array of processors) is shown in

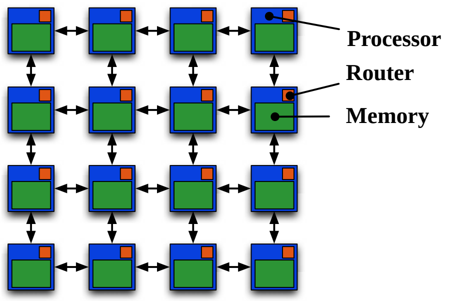

Figure 4. Parallel chip multiprocessor showing a 4×4 configuration.

Figure 4. The arrows correspond to connections between the routers in each processor/router/memory tile, and any processor can send a message to any other processor via a multi-hop route through the routers as in conventional multiprocessor systems [18].

Individual processing elements in the parallel array are entirely asynchronous, as is the interconnection network. Figure 5 shows the microarchitecture of the asynchronous processor used in each tile. The processor has an activity-driven architecture that is designed around a hardware structure called the event queue. Initially, this event queue is empty. An incoming message notification is detected by dedicated hardware and is converted into a token that is inserted into this event queue. When the processor receives a token from the event queue, it wakes up and begins instruction execution from an address specified in a hardware table. These instructions constitute the event handler. Once the event handler is complete, a special instruction is executed that makes the processor once again wait for a new token in the event queue.

Implementation. To implement acyclic Δ-dataflow networks on such a platform is relatively straightforward. The functions f and g are specified for each compute node and implemented using the C programming language. Our compiler will generate the appropriate machine code for

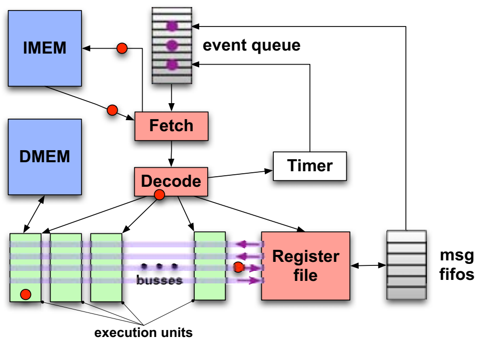

Figure 5. Component processor in parallel architecture.

the processor. Edges in the graph where updates are propagated correspond to messages being sent between the corresponding processors in the NoC array. The event queue structure in each processor provides the necessary mechanism to activate the processor and execute the appropriate code in response to the incoming update.

To illustrate this, we use the functions f and g from Example 2. The number of instructions that would be executed for the addition and averaging operation is twelve instructions when a new input arrives but the output remains unchanged (case 1), and seventeen instructions when a new value must be propagated to the output (case 2). We combine these instruction count estimates with results from SPICE simulations to estimate the power and performance of the NoC processors [8, 11]. Operating at the nominal voltage (1.8V) for high-performance operation, executing the averaging node will require 2.7nJ and 3.9nJ of energy for the two cases respectively, and the execution will require 50ns and 71ns respectively. At low voltage (0.6V) for low power operation, these operations will require 0.26nJ and 0.37nJ respectively, while taking $0.5\mu s$ and $0.7\mu s$ for execution. For comparison purposes, conventional low-power microprocessors like the Intel XScale require about 1nJ per instruction [14]. Implementing Δ-dataflow networks on conventional platforms not only requires more energy per basic operation, but the scheduling (handled by the event queue) would have to be implemented in software as well, further increasing the implementation overhead.

5 Summary

We presented a new class of dataflow-like networks suitable for efficient event-processing in sense-and-respond systems. The networks have a built-in notion of filtering, because they only propagate changes and suppress their computation whenever the same value is being produced. Sensors or event-streams can flow through these networks, and the computation nodes process these streams to produce new event streams that can be eventually used to detect complex conditions. We showed several properties of these networks by considering a concrete model of the network components. We also described how these event-processing networks can be efficiently mapped to an asynchronous VLSI implementation, although other implementations are also possible.

References

[1] Arvind and D.E. Culler. Dataflow Architectures. MIT LCS Technical Memo MIT/LCS/TM-294, February 1996.

[2] R.M. Butler and E.W. Lusk. Monitors, messages, and clusters—the p4 parallel programming system. *Parallel Computing*, **20**(4):547–564, 1994.

[3] K. Mani Chandy, Brian E. Aydemir, Elliott Karpilovsky, Dan Zimmerman. Event-Driven Architectures for Distributed Crisis Management. *Proceedings 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2003.

[4] K. Mani Chandy, Brian E. Aydemir, Elliott Karpilovsky, Dan Zimmerman. Event Webs for Crisis Management. *Proceedings 2nd IASTED International Conference on Communications, Internet and Information Technology*, November 2003.

[5] K. Mani Chandy, Adam Rifkin, and Eve Schooler. Using Announce-Listen with Global Events to Develop Distributed Control Systems. *Concurrency: Practice and Experience*, 1998.

[6] D.D. Clarke. The Design Philosophy of the DARPA Internet Protocols. *Proceeding of SIGCOMM 1988*.

[7] J.B. Dennis. Data Flow Supercomputers. *Computer*, **13**(11):48–56, November 1980.

[8] V. Ekanayake, C. Kelly, and R. Manohar. An Ultra Low Power Processor for Sensor Networks. *Proceedings of the 11th International Symposium on Architectural Support for Programming Languages and Operating Systems*, November 2004.

[9] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 1982, pp. 17–37.

[10] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Proceedings of the IFIP Congress 74*, pp. 471–475, 1974.

[11] C. Kelly, V. Ekanayake, and R. Manohar. SNAP: A Sensor-Network Asynchronous Processor. *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, pp. 24–33, May 2003.

[12] C. Kelly and R. Manohar. An Event-Synchronization Protocol for Parallel Simulation of Large-Scale Wireless Networks. *Proc. 7th International Symposium on Distributed Simulation and Real-Time Applications*, October 2003.

[13] L.J. Guibas, H.T. Kung, and C.D. Thompson. Direct VLSI Implementation of Combinatorial Algorithms, *Proc. Caltech Conference on VLSI*, pp. 509–525, January 1979.

[14] Intel PXA255 XScale Processor Datasheet. Available from http://www.intel.com/

[15] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[16] Message Passing Interface Forum. MPI: A message passing interface standard, May 1994.

[17] C.L. Seitz. System Timing. In *Introduction to VLSI Systems* by C.A. Mead and L. Conway, (Addison-Wesley, 1979).

[18] C.L. Seitz. Mosaic C: An Experimental Fine-Grain Multicomputer. Proceedings of the International Conference on Future Tendencies in Computer Science, Control, and Applied Mathematics, Lecture Notes in Computer Science, 1992.

[19] K. Yeager. The MIPS R1000 Superscalar Microprocessor. *IEEE Micro*, **16**(2):28–40, April 1996.