Dataflow architecture

Are dataflow computers commercially viable?

Krishna M. Kavi and Behrooz Shirazi

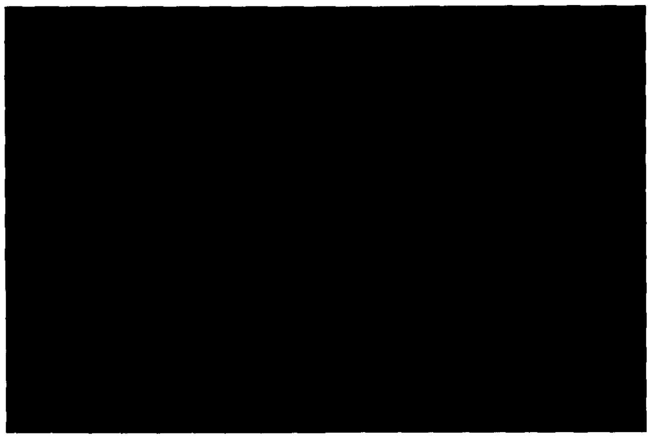
IMAGE BANK

Two fundamental issues that multi-processing must deal with are synchronization and latency. Synchronization is required to coordinate dependent tasks in order to assure deterministic execution. Task dependence is either due to data or control dependencies. Data dependencies arise due to shared variables (including flow, anti and output dependences), while control dependencies define the order of execution. Flow dependence arises when a computation writes a variable that is read by a subsequent operation; antidependence occurs when one operation reads a variable that is written by a subsequent operation, while output dependence occurs when two computations write to the same variable. Latency is defined as the time to complete an operation. Latency often implies delays due to sharing of resources such as memory and communication subsystems. These delays can be minimized by properly partitioning and distributing resources among the processing elements.

Consider the following program segment that computes the inner product of two arrays.

```
        SUM = 0.0
        DO 100 I= 1 to N
          SUM = SUM + A(I) * B(I)
100     CONTINUE
```

When the loop iterations are executed concurrently on a conventional multiprocessor system, access to SUM must be synchronized. Memory latencies resulting from conflicts in accessing the arrays A and B can be minimized by properly "slicing" the arrays and distributing the memory across the multiple processing elements. This problem of data distribution is a non-trivial task, as an optimal partition depends not only on the multiprocessor organization but also on the access patterns of the algorithm itself.

In data flow model of computation, the data (or availability of data) determines which instruction is executed next, unlike the conventional von Neumann mode that prescribes sequential execution. Any instruction can be executed as soon as all the operands for that operation are available, and there is no program counter that defines instruction sequencing. This *data-driven* execution can lead to exploitation of inherent fine-grain parallelism. Implicit in the model is the lack of variables for data - only data values flow among computations. Lack of mutable variables (freedom from side-effects) eliminates anti and output dependencies. Synchronization is defined exclusively by the flow (data) dependencies.

Fig. 1 shows the dataflow graph for the body of the loop in the inner product program example. Since dataflow model has no concept of a mutable variable storage, the assignment to SUM inside the loop body must be defined as the creation of new SUM values.

next SUM = SUM + A(I) * B(I).

The "synchronization" among the

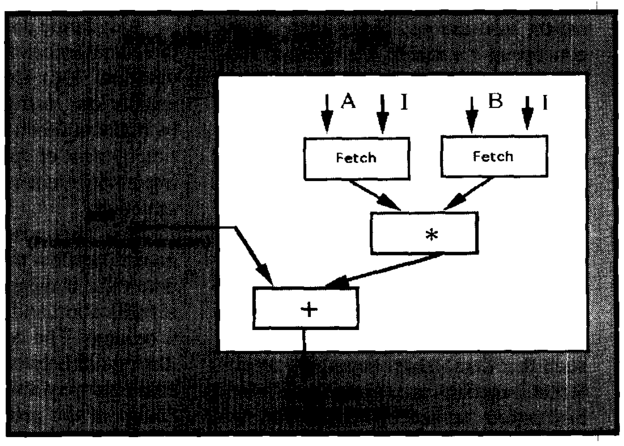

Fig. 1

Operation Code	
Avail	Operand - 1
Avail	Operand - 2
Destination - 1	
Destination - 2	

Fig. 2

loop iterations is achieved by the data dependencies forced by the "next SUM" value. A loop iteration is not complete until the value of SUM is provided by a previous iteration. This can be viewed as a latency - delay in accessing the value of SUM. In Fig. 1, the operation Fetch is used to access arrays. Data structure presents challenges in dataflow computers, as will be discussed later.

Static dataflow computers

Since dataflow computers have a data-driven organization, it is possible to design a dataflow processor with three cycles.

1. ***Examine.*** Examine each instruction to determine the availability of its operands.

2. ***Select.*** Schedule a subset of enabled instructions. Often, more instructions are enabled than available processing resources.

3. ***Update.*** When enabled instructions are complete, the results are used to update the availability of operands for other instructions.

Let us examine an instruction format for such a machine. The instruction *(instruction packet)* should include the operation code, a place for each of its operands, and the addresses of the destination instructions. Since there is no concept of a variable, the operand values are stored directly in the instruction (similar to "immediate" or "literal" operands). Fig. 2 shows a simple example of an instruction packet with two operands and two destinations. The Avail flag indicates if the operand value is available.

In order to execute all loop iterations of the inner product example concurrently, we must make copies of the instructions constituting a loop iteration, such that data values belonging to different iterations are stored in different instruction packets. Alternatively, we can limit the parallelism by allowing only one loop iteration to be active. This is the solution used in static dataflow computers. In other words, static dataflow machines allow at most one data token on an arc of a dataflow graph. Such a restriction permits the use of "control tokens" to synchronize concurrent computations (such as loop iterations).

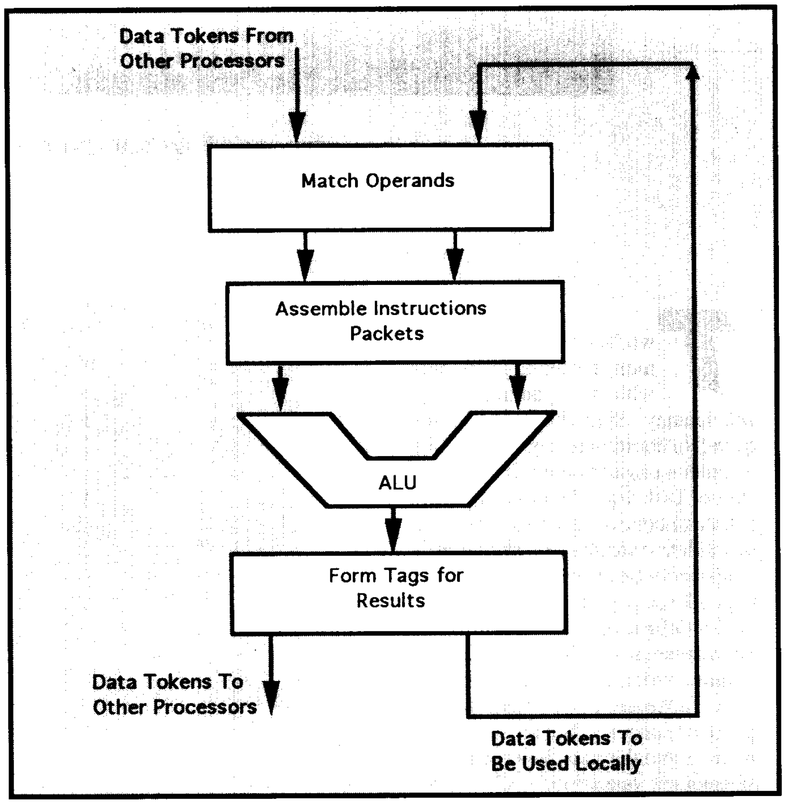

Fig. 3

Dynamic dataflow computers

A dynamic dataflow model permits the coexistence of multiple tokens on a dataflow arc. Conceptually, we can modify our instruction packet (Fig. 2) by replacing each operand to an instruction by a list of operand values, belonging to different invocations of the instruction.

It is also possible to design a dynamic dataflow model as follows. Each data value is tagged with the instruction identification, and the "context" to which it belongs. The context can be used to distinguish between loop iterations. The data tags may also include the identification of the processing element in a multiprocessor environment. Since all operands of an instruction will have the same tags, an instruction is enabled when all such data tokens are received. The processor for such a computer system must perform the following functions.

1. *Match.* As data tokens are received, the tags are used to match operands that belong to the same activation of an instruction.

2. *Execute.* Enabled instructions are scheduled on the ALUs.

3. *Output.* The result values must be tagged with the destination instruction and context information.

If we restrict ourselves to at most two operands per instruction, the matching operation can easily be implemented using an associative memory, where the data tags are used in the associative search. Initially all locations are flagged as *empty*. When one of a matching pair of data items arrives, the data value is stored in the associative memory, and the memory location is flagged *full.* When the second matching data item arrives and notices the full flag an instruction packet comprised of the new data item, the first data item (retrieved from the associative memory), the operation code, and information to generate tags for result data items are sent to the ALU. Fig. 3 shows a pipelined processing unit that can be used in a dynamic dataflow computer.

Instructions with more than two operands can be represented as multiple two-operand instructions. For example, a three-operand instruction can be represented as two two-operand instructions. One operand instructions (or two-operand instructions where one operand is a constant) present no difficulties to the matching unit.

Since capacious, fully associative memories are not readily realizable, alternatives must be found for matching tagged data operands. One such alternative is used in the Monsoon dataflow computer system. The tags of data tokens are used as "memory" addresses. Each memory location is flagged with an empty/full bit. Conceptually, data items generated by the ALU are stored using the tag as an address into a global memory. In the case of two operand instructions, the empty/full bit can be used to determine the availability of one or both operands.

Let us examine the instruction packets of a Monsoon-like dataflow processor. The instruction packet will contain the operation code, the data values of the operands, and one or more destinations (see Fig. 2). Unlike the static dataflow, Monsoon instruction packet must contain sufficient information to create the tags with the result of the operation. Some results are destined to an instruction within the same context, while some are destined to other contexts. Unlike conventional machines, the instruction fetch is initiated when the operands for the instruction are available. In case of two-operand instructions, when the second operand discovers that the "memory" location specified by its tag is already full, an instruction fetch is initiated.

Let us examine data tags more closely. Each tag consists of a frame and an instruction part. The instruction part is used in fetching the instruction once the operands are available. The frame is defined by a context (C) field and an offset within the context (S). Typically, a context implies a loop iteration, a code block, or a function invocation. In order to limit the size of the tag field, the context tags must be carefully recycled. This can be achieved by restricting the number of loop iterations that can be executed in parallel. The number of contexts can also be reduced by increasing the size of a context. Assuming that a context is assigned to a single processor, either of the two solutions result in reducing the amount of concurrency.

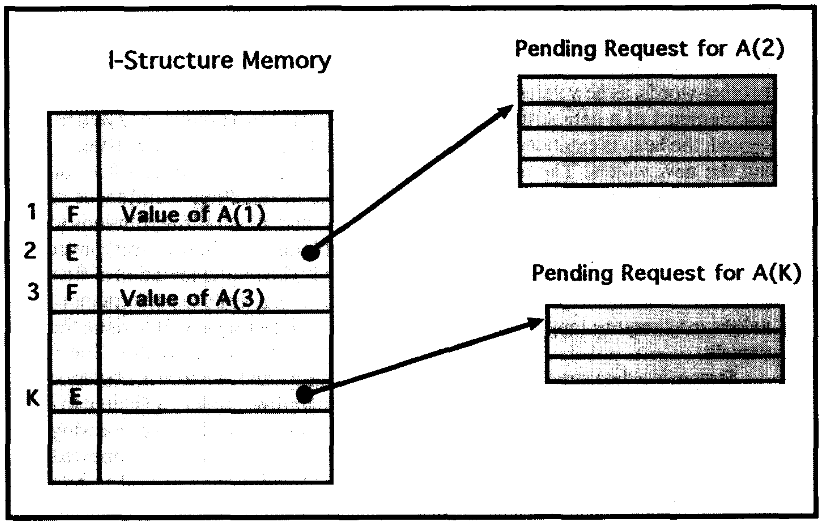

Fig. 4

When the size of a context (i.e., the C field of a data tag) is small, the matching of operands can be performed using register files. Each context will be assigned a set of registers, and the S field of a data tag addresses a specific register in its context. Using empty/full bits with registers, operands for instructions within a context can be matched. If multiple contexts are assigned to the same processor, cache memories can be used instead of register files.

Handling data structures in dataflow environments

So far we have paid very little attention to how aggregate data values can be handled. In our discussions of data tags we ignored the type of value carried in tokens. Preservation of dataflow semantics implies that, when dealing with structured data types like arrays, the entire data structure must be carried in a token. This is because modification to even one element of an array results in a new data value. Obviously, this is inefficient. On the other hand, if the token contains a pointer to the structure, care should be exercised such that no elements of the structure are modified (a value is created only once, and any further assignments must be flagged as errors), or new structures are dynamically created with any modifications. The data driven model implies that any attempt to access elements of the structure that are yet to be generated should be forced to wait.

An I-structure is a special kind of array designed to handle arrays in dynamic dataflow computers. Three operations are defined with I-structures: *allocate, i-store, ifetch.* The *allocate (A, N)* returns an N-element empty array (i.e., each element of the structure is flagged as empty). Each element of an array can itself be an I-structure; multidimensional arrays can thus be specified. An element of I-structure can be assigned a value V no more than once using *i-store(A, I, V).* The I-th element of the array A is now set to full. Any attempt to store values into a full element results in error. An element of the array can be accessed using *i-fetch (A, I).* If the I-th element is already defined (indicated by the full status), the value of the element is returned. Otherwise, the request is deferred until the value is available. Fig. 4 shows an I-structure example with pending requests for unavailable array elements.

A heap structure can be used to handle non-homogeneous data structures. A heap can be considered as a a tree where each node is either a data value or a pointer to a data value. Each element of the data structure is represented by a node in the heap; nodes are addressed by their relative positions in the data structure. When an element of the data structure is modified, the corresponding node of the heap is modified to point to a newly created node containing the value. In other words, as new values to individual elements of a data structure are assigned, the heap is extended by appending the new values. The major advantage of this implementation is in terms of the memory utilization; unmodified elements are not copied when some elements of the data structure are assigned new values. However, access to values may require lengthy pointer traversals.

Stream is a data structure that is very suitable for a dataflow model of computation because of the limited access and the possibility of eager evaluation. Elements of stream can only be accessed sequentially. New elements can be added (created) either at the head or the tail of an existing stream. CAR returns the first element of the stream, while CDR returns the remaining stream after deleting the first element. Eager evaluation is possible because the consumer of a stream does not have to wait until the entire structure is generated; stream elements can be consumed as they are generated.

Summary and prognosis

The dataflow model of computation is neither based on memory structures that require inherent state transitions, nor does it depend on history sensitivity. This "purity" permits the use of the model to represent maximum concurrency to the finest granularity, and facilitates dependency analysis among computations. The authors of this paper have used the model as a formalism for the specification and analysis of concurrent processing systems. Optimizing compilers convert programs written in languages such as FORTRAN into dataflow graphs (usually known as directed acyclic graphs or DAGs), for the purpose of analyzing data dependencies. Lately, assignments to a shared variable in a multiprocessing environment are transformed to "single-assignment" dataflow-like variables for the purpose of increasing the available parallelism.

Despite the advantages, there are no commercially successful implementations of the data-driven model. Critiques of dataflow argue that functional semantics and freedom from side effects are nice for the programmer, but well-known compiler techniques applied to languages like FORTRAN allow equal exploitation of parallelism. They point out the importance of memory hierarchy, memory management and garbage collection in a system in which memory is not used in the conventional sense. They claim that the dataflow model is weak in handling recurrences and arrays, and all dataflow solutions to these problems (e.g., I-structures) are complicated. The implementation of fine-grain parallelism leads to performance penalties in detecting and managing the high-level of parallel activities. The lack of memory (and memory hierarchy) makes the dataflow model inefficient to implement on register-based processing units.

There is, however, a renewed interest in the design of hybrid architectures that combine dataflow and von Neumann models of computation. Although, classical dataflow model emphasizes fine-grain parallelism, program partitioning techniques can be used to produce medium to coarse grain parallelism. Each macro-dataflow node can be treated as a context or frame, as in the "Monsoon" processor. Coarse grain dataflow can also help in utilizing techniques for improving processing speeds such as instruction look-ahead, pipelines and cache memories.

Traditional compiler optimizations can be applied when dealing with macro-dataflow programs. For example, even though dataflow model requires the creation of a new array each time array elements are modified, if all array updates are local to a macro dataflow node, the compiler can suppress the creation of new arrays, and permit in-place updates (as done in imperative languages). Sisal, a dataflow language developed at Lawrence Livermore Laboratories, relies on in-place updates. Likewise, registers can be used to hold intermediate or temporary values for a macro dataflow node.

As demonstrated by the Monsoon system, dataflow systems can be designed using traditional register files and activation contexts. However, the current implementation of the Monsoon architecture is extremely complex in terms of the number and type of instructions (more than 100 instructions that fall into 26 different encoding classes), the amount of instruction decoding needed (two level decoding using five different tables), and in its I-structure support. As yet unanswered is the question of the effectiveness of the hybrid architectures for conventional (imperative) parallel programming model. We believe that when traditional compiler techniques are utilized, Monsoon-like dataflow computers will compete very favorably with RISC-like processor, particularly for scientific applications.

Read more about it

• A.J. Anderson. *Multiple Processing: A systems overview,* Prentice Hall International (UK), Ltd., 1989. Chapter 10 of this book contains an extensive treatment of the dataflow model, and some dataflow computers. It also includes a good discussion on the advantages and disadvantages of the dataflow model.

• Arvind and D.E. Culler. "Dataflow Architectures", In *Annual Reviews in Computer Science, Vol. 1,* pp 225-253, Annual Reviews Inc., Palo Alto, CA, 1986. This paper describes dynamic dataflow architectures and I-structures.

• D.D. Gajski, D.A. Padua, D.J. Kuck and R.H. Kuhn. A second opinion on dataflow machines and languages, *IEEE Computer, Vol.* 15, No. 2, Feb. 1982, pp 58-69. Presents a critical view of the dataflow model, and claims that compiler techniques for imperative languages are sufficient to exploit available parallelism.

• G.M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor,* MIT Press, Cambridge, MA 1991. Describes the details of the Monsoon dataflow processor.

• P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, Data driven and demand driven computer architecture, *ACM Computing Surveys,* Vol. 14, No. 1, March 1982, pp 93143. Although dated, it is an excellent survey of dataflow computer systems.

About the authors

Krishna M. Kavi is a professor at the University of Texas at Arlington. He is an IEEE CS distinguished visitor, an editor of the IEEE CS Press, and a senior member of the IEEE. He extended the dataflow model of computation for use in specification and analysis of parallel processing system. His other research interests include object-based computer systems and real-time formalisms.

Behrooz Shirazi is an associate professor at the University of Texas at Arlington. He has been investigating approaches for efficient handling of data structures in dataflow processing environments. His other research interests include scheduling and load balancing in parallel processing systems. ■