D

DATAFLOW COMPUTERS: THEIR HISTORY AND FUTURE

INTRODUCTION AND MOTIVATION

As we approach the technological limitations, concurrency will become the major path to increase the computational speed of computers. Conventional parallel/concurrent systems are based mainly on the control-flow paradigm, where a primitive set of operations are performed sequentially on data stored in some storage device. Concurrency in conventional systems is based on instruction level parallelism (ILP), data level parallelism (DLP), and/or thread level parallelism (TLP). These parallelisms are achieved using techniques such as deep pipelining, out-of-order execution, speculative execution, and multithreaded execution of instructions with considerable hardware and software resources.

The dataflow model of computation offers an attractive alternative to control flow in extracting parallelism from programs. The execution of a dataflow instruction is based on the availability of its operand(s); hence, the synchronization of parallel activities is implicit in the dataflow model. Instructions in the dataflow model do not impose any constraints on sequencing except for the data dependencies in the program. The potential for elegant representation of concurrency led to considerable interest in dataflow model over the past three decades. These efforts have led to successively more elaborate architectural implementations of the model. However, studies from past projects have revealed a number of inefficiencies in dataflow computing: the dataflow model incurs more overhead during an instruction cycle compared with its control-flow counterpart, the detection of enabled instructions and the construction of result tokens generally will result in poor performance for applications with low degrees of parallelism, and the execution of an instruction involves consuming tokens on the input arcs and *generating* result token(s) at the output arc(s), which involves communication of tokens among instructions. Recent advances that may address these deficiencies have generated a renewed interest in dataflow. In this article we will survey the various issues and the developments in dataflow computing.

This chapter is organized as follows: the Dataflow Principles section reviews the basic principles of the dataflow model. The discussion includes languages supporting dataflow model. The Dataflow Architectures section provides a general description of the dataflow architecture. The discussion includes a comparison of the architectural characteristics and the evolutionary improvements in dataflow computing, including pioneering pure dataflow architectures, hybrid architectures attempting to overcome the shortcoming of pure dataflow systems, and recent attempts to improve the hybrid systems. The next section outlines research issues in handling data structures, program allocation, and application of cache memories. Several proposed methodologies will be presented and analyzed. Finally, the last section concludes the article.

DATAFLOW PRINCIPLES

The dataflow model of computation deviates from the conventional control-flow method in two fundamental ways: asynchrony and functionality. Dataflow instructions are enabled for execution when all the required operands are available, in contrast to control-flow instructions, which are executed sequentially under the control of a program counter. In dataflow, any two enabled instructions do not interfere with each other and thus can be executed in any order, or even concurrently. In a dataflow environment, conventional concepts such as "variables" and "memory updating" are nonexistent. Instead, objects (data structures or scalar values) are consumed by an actor (instruction) that yields a result object that is passed to the next actor(s). It should be noted that some dataflow languages and architectures, however, use variables and memory locations for the purposes of convenience and of efficiency.

Dataflow Graphs

Dataflow graphs can be viewed as the machine language for dataflow computers. A dataflow graph is a directed graph, $G(N, A)$, where nodes (or actors) in N represent instructions and arcs in A represent data dependencies among the nodes. The operands are conveyed from one node to another in data packets called tokens. The basic primitives of the dataflow graph are shown in Fig. 1. A data value is produced by an operator as a result of some operation f. A true or false control value is generated by a decider (a predicate), depending on its input tokens. Data values are directed by means of either a switch or a merge actor. A switch actor directs an input data token to one of its outputs, depending on the control input. A Merge actor passes one of its input tokens to the output based on the value of the control token. Finally, a copy is an identity operator which duplicates input tokens. Figure 2 depicts the dataflow graph of the following expression:

$$sum = \sum_{i=1}^{\mathrm{N}} f(i)$$

Note the elegance and flexibility of the dataflow graph to describe parallel computation. In this example, the implicit parallelism within an iteration is exposed. Furthermore,

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

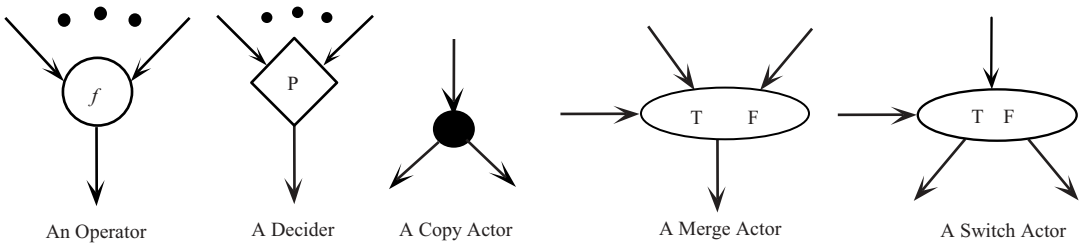

Figure 1. Basic primitives of the dataflow graph.

because of the functional properties of operations, the function f can be invoked simultaneously for all values of i. Thus, given sufficient amount of resources, N iterations of function f can be executed concurrently.

Dataflow Languages

Any dataflow language should permit the specification of programs that observe dataflow principles. In terms of programming language semantics, these principles translate into freedom from side-effects (prohibit modification of variables either directly or indirectly), single assignment (values associated with variables cannot be modified), and locality of effect (instructions do not have unnecessary far-reaching data dependencies). In this section we introduce three dataflow languages that received considerable attention in the literature.

VAL: A Value-oriented Algorithmic Language. VAL is a high level programming language developed at MIT (1), and can be viewed as a textual representation of dataflow graphs. VAL relies on pure functional language semantics to exploit implicit concurrency. Since dataflow languages use single assignment semantics, the implementation and the use of arrays present unique challenges (see Research Issues). In VAL, array bounds are not part of the type declarations. Operations are provided to find the range of indices for the declared array. Array construction in VAL is also unusual to improve concurrency in handling arrays. It should be noted that because we must maintain single assignment feature of functional languages, traditional language syntax to accumulate values (for example, the sum in Fig. 2) need some changes. To express such concurrencies, VAL provides parallel expressions in the form of *forall*. Consider the following examples:

1. forall i in [array_liml(a), array_limh(a)]

```
a[i] := f(i);
```

2. forall i in [array_liml(a), array_limh(a)]

```
eval plus a[i];
```

If one applies imperative semantics, both examples proceed sequentially. In the first case, the elements of the array a are constructed sequentially by calling the function f with different values of the index i. In the second example, we compute a single value that represents the sum of the elements of the array a, which represents sequential accumulation of the result. In VAL, the construction of the array elements in example 1 can proceed in parallel because all functions in VAL are side-effect free. Likewise, the accumulation in example 2 also exploits some concurrency because VAL translates such accumulations into a binary tree evaluation.

In addition to loops, VAL provides sequencing operations, *if-then-else* and *tagcase* expressions. When dealing with one of data type, tagcase provides a means of interrogating values with the discriminating unions.

VAL did not provide good support for input/output operation nor for recursion. These limitations allowed for a straightforward translation of programs to dataflow architectures, particularly static dataflow machines (see the earlier Dataflow Architectures section). The dynamic features of VAL can be translated easily if the machine supported dynamic graphs, such as the dynamic dataflow architectures.

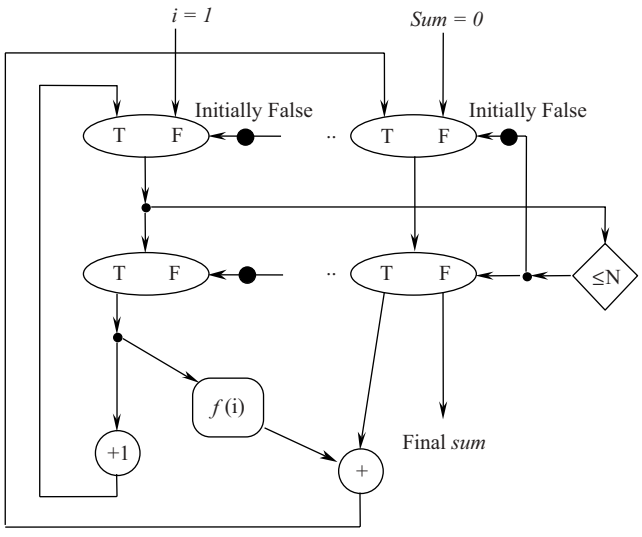

Figure 2. A dataflow graph representation of $sum = \sum_{i=1}^{N} f(i)$.

Id: Irvine Dataflow language. Id is a dataflow language that originated at the University of California-Irvine (2), and was designed to permit high-level programming language for the dynamic dataflow architecture proposed by Arvind (see the Earlier Dataflow Architectures section). Id is a block-structured, expression-oriented, single assignment language. An interpreter was designed to execute Id programs on dynamic dataflow architectures. Data types in Id are associated with values, and variables are typed implicitly by the values they carry. Structures include both arrays and (record) structures; and elements can be accessed using either integer indices or string values that define the name of the element (for example, t["height"]). Structures are defined with two operators: select and append. Select is used to get the value of an element, whereas append is used to define a new structure by copying the elements of the original structure and adding new values defined by the append operation.

Id programs consist of side-effect free expressions and expressions (or subexpressions) can be executed in any order or concurrently based on the availability of input data. Loops in Id can be understood easily from the following example, which computes $\sum_{i=1}^{N} f(i)$

```
(initial i ← 1; sum ← 0;
while i ≤ N do new i ← i+1;
         new sum ← sum + f(i);
         return sum)
```

Id uses the concept of "new" to define a new value associated with an expression. It should be noted that a variable is not assigned a new value (like in conventional languages), but a new value is generated – variables are used only for the convenience of writing programs. It is also convenient to remember that the expressions in a loop can form recurrence expressions.

Procedures and functions in Id are pure functions and represent value(s) returned by the application of the function on the input values. Recursive procedures can be defined by associating names with procedure declarations. For example:

$$\text{y} \leftarrow \text{procedure f(n)(if n} = 0 \text{ then 1else n*f(n-1))}$$

defines factorial recursively, and we can invoke the procedure, for example as y(3).

Because no translators to convert Id programs to conventional (control-flow) architectures were developed, Id was used mostly by those with access to dynamic dataflow processors and to Id interpreters.

SISAL: Streams and Iterations in a Single Assignment Language. Sisal is the best-known dataflow language, mostly because of the support provided by the designers. Sisal received a fairly wide acceptance during the 1990s, because Sisal compilers generated optimized C as their intermediate representations and thus could be run on any platform with a C compiler. Although it is not as widely known now, Sisal translator and run-time support software are still available for Unix based systems and can be obtained from the web at http://sisal.sourceforge.net/. Sisal 2.0 provided multitasking (or multithreading) to support dataflow-style parallelism on conventional shared memory multiprocessors (4).

Sisal programs consist of one or more separately compilable units, which include a simple program, modules, and interfaces. A module is similar to a program but is not a starting point of execution. It pairs with an interface to export some of its types and function names. Like Id, Sisal supports scalar data types and structures (records, union, arrays, and streams). A stream is a sequence of values produced in order by one expression (thus it consists of homogeneous typed values), and is consumed in the same order by one or more other expressions. Sisal permits the creation of new values (and associates them with the same name).

```
for i := 1;
while ( i <5) do
         new i := i+2;
         j := i + new i;
returns product (i+j)
end for
```

This program constructs implicitly a stream of values inside the loop and returns the product of the elements of the stream. The values of the stream are the values of (i + j): 7, 13. Thus 91 is returned by the loop.

Sisal expressions can loop over the elements of an array (called array scattering) or over the elements of a stream (stream scattering) (5). As with VAL, Sisal can perform reduction operations concurrently using binary tree evaluations. Sisal has predefined reduction operations to evaluate sum, product, min, and max of a set of values. Catenate is a reduction that returns the concatenation of a sequence of one-dimensional array or a stream. Consider:

```
for i in[ 1..N] do
         return sum f(i);
```

which uses sum as a reduction operation to produce $\sum_{i=1}^{N} f(i)$.

Additional reduction functions can be defined by the programmer. Consider the following example to build histograms.

```
reduction histo(v, N: integer returns array of integer)
initial
         hacc := array[0..N+1:0];
 in

         idx := if(v <1) then 0
                  elseif (v > N) then n+1
                  else v
                  end if;
         new hacc := hacc[idx: hacc[ idx] +1]
returns hacc
end reduction
```

Sisal's popularity also is caused by the concept of modules and interfaces: The interface shows the function templates that are visible publicly and the module defines the implementations of the functions. Sisal implementations also permit foreign code (written in a different language), by associating an interface with the foreign

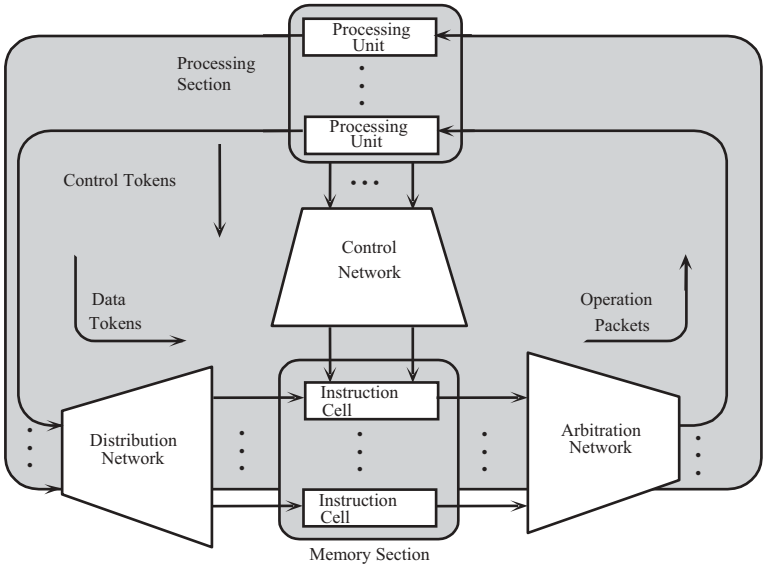

Figure 3. The basic organization of the static dataflow model.

code. Consider for example to import MathLib functions to Sisal programs:

```
interface MathLib in FORTRAN
          function sin (x: real returns real);
          function tan (x: real returns real);
end interface
```

DATAFLOW ARCHITECTURES

Architectural implementations of dataflow traditionally have been classified as either static or dynamic. The static approach allows at most one instance of a node to be enabled for firing, i.e., a dataflow actor can be executed only when all of its input tokens are available and no tokens exist on any of its output arcs. On the other hand, the dynamic approach permits simultaneous activation of several instances of a node during the run-time by viewing arcs as buffers containing multiple data items. To distinguish between different instances of a node (and routing data for different instantiations of the node), a tag is associated with each token that identifies the context in which a particular token was generated. An actor is considered executable when all of its input tokens with identical tags are available.

The static dataflow model has a simplified inherent mechanism to detect enabled nodes, but the model limits the performance because iterations are executed one at a time. The dynamic dataflow allows greater exploitation of parallelism; however, this advantage comes at the expense of the overhead in terms of the generation of tags, larger data tokens, and complexity of the matching tokens. A more subtle problem with the token matching is the complexity involved in allocation of resources (i.e., memory cells). A failure to find a match implicitly allocates memory within the matching hardware. If the matching unit becomes overcommitted, the program may deadlock.

Dataflow architectures have also been classified as pure dataflow architectures, macro dataflow architectures, and hybrid dataflow architectures. Detailed discussion about this classification is beyond the scope of this article and interested reader is referred to Ref. 6.

Earlier Dataflow Architectures

This section discusses three classic dataflow machines: the static dataflow machine, the (dynamic) manchester machine, and the explicit token store. These projects represent the pioneering work in the area of dataflow. The foundation they provide has inspired many other dataflow projects.

Static Model. The general organization of the original (static) dataflow machine is depicted in Fig. 3 (Table 1) (7). The memory section is a collection of memory cells, each cell composed of three memory words that represent an instruction template. The first word of each instruction cell contains op-code and destination address(es), and the next two words represent the operands. The design has envisioned six types of templates that represent binary and unary operators, binary and unary deciders (predicates), and binary and unary Boolean operators. The processing section is composed of five pipelined functional units, which perform the operations, form the result packet(s), and send the result token(s) to the memory section. The arbitration network is intended to establish a smooth flow of enabled instructions (i.e., instruction packet) from the memory section to the processing section. An instruction packet contains the corresponding op-code, operand value(s), and destination address(es). The distribution network is intended to transfer the result packets from the processing section to the memory section. Finally, the control network in designed to reduce the load on the distribution network by transferring the Boolean tokens and the acknowledgement signals from the processing section to the memory section.

Figure 4 shows the dataflow graph and the contents of the memory section for the $Y(t) = A^*X(t) + B^*Y(t\text{-}1)$

Table 1. Earlier Dataflow Architectures

Name	Country	Type
MIT Static Dataflow (1975) (7)	USA	Static
Manchester Dataflow (1977) (8)	England	Dynamic
MIT Tagged Token (1978) (9)	USA	Dynamic
CSIRAC II (1978) (9)	Australia	Dynamic
DDM1 Utah Data Driven (1978) (10)	USA	Static
LAU System (1979) (9)	France	Static
TI Distributed Data Processor (1979) (11)	USA	Static
NEC Image Pipelined Processor (1980) (12)	Japan	Static
NTT Dataflow Processor Array (1983) (13)	Japan	Dynamic
Distributed Data Driven Processor (1983) (14)	Japan	Dynamic
Stateless Dataflow Architecture (1983) (15)	England	Dynamic
SIGMA-1 (1984) (16)	Japan	Dynamic
Parallel Inference Machine (1984) (17)	Japan	Dynamic

$+C^{*}Y(t\text{-}2)$. For the sake of simplicity, representation details in the memory are omitted. Note that each memory cell is numbered to correspond to the node number in the dataflow graph.

Manchester Dynamic Model. Figure 5 shows the block diagram of the *dynamic dataflow* system prototyped at the Manchester University (Table 1). It is designed as a back-end, composed of five units organized as a pipeline ring: The switch unit establishes communication between the front-end and back-end processor, and routes the result tokens back to the pipeline ring. The token queue is a First-in-first-out buffer that stores temporarily tokens traversing on the data-flow graph arcs. The basic operation of the matching unit is to bring together tokens with identical tags by pairing associatively tokens with the same destination node address and context. The dataflow program that represents the code for an operation is stored in the node store. The processing unit, a micro-programmed, 2-stage pipeline unit, executes the dataflow operations. The first stage handles the generation of result tokens and the association of tags with tokens. The second pipeline stage consists of 15 functional units to perform the necessary operations.

Explicit Token Store. Despite the potential parallelism promised by the dynamic dataflow model, early experiences have identified the following shortcomings in implementing the model:

- Overhead involved in matching tokens (and the need for associative matching),
- Complex resource allocation,
- The inefficiency of the dataflow instruction cycle (token driven models require multiple cycles through the pipeline to complete execution), and
- Nontrivial mechanisms to handle data structures (see the Research Issues section).

Performance of the dynamic dataflow architecture is related directly to the rate at which the matching unit operates. To facilitate this process while considering the cost, Arvind proposed a pseudo-associative matching mechanism that requires typically several memory accesses (9). A failure in finding a match implicitly allocates

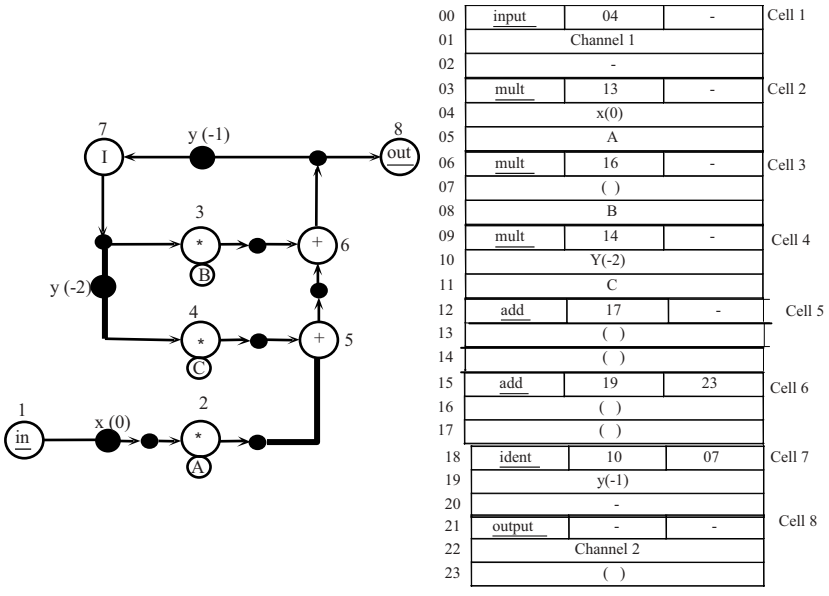

Figure 4. A dataflow graph and its representation (MIT Static Model).

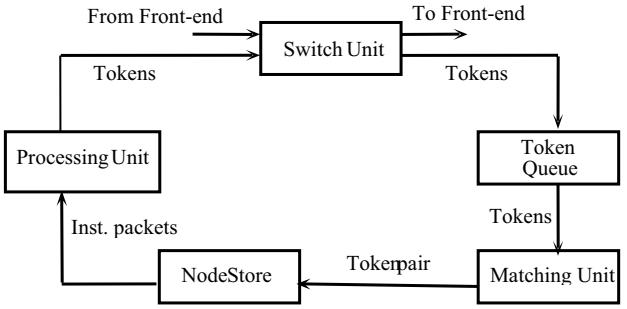

Figure 5. The Manchester Dynamic dataflow Machine.

memory within the matching unit — mapping a code-block to a processor places an unspecified commitment on the processor's matching unit. This can result in a deadlock if this resource becomes overcommitted. Another problem with dataflow processing is caused by the duration of the instruction cycle relative to its control-flow counterpart.

To overcome the inefficient matching of dynamic dataflow model, explicit token store (ETS) proposed a direct matching (9). Storage (called *activation frames*) is allocated dynamically for all the tokens that can be generated by a code block (a code block represents a function or a loop iteration). The usage of memory locations within the activation frame is determined at compile time; however the allocation of storage is determined at run time. A computation is described completely by an instruction pointer (IP) and an activation frame pointer (FP) and the pair <FP. IP> called a *continuation*. A typical instruction specifies an opcode, an offset in the activation frame where a match for its inputs will take place, and one or more displacements that define the destination instructions that will receive the result token(s). Each displacement is also accompanied by an input port (left/right) indicator that specifies the appropriate input arc for a destination actor. Figure 6 shows an example of the ETS code block invocation and its corresponding instruction and frame memory. When a token arrives at an actor (for example, Add), the IP part of the continuation points to the instruction that contains an offset *r* as well as displacements for the destination instructions. The system achieves the actual matching process by

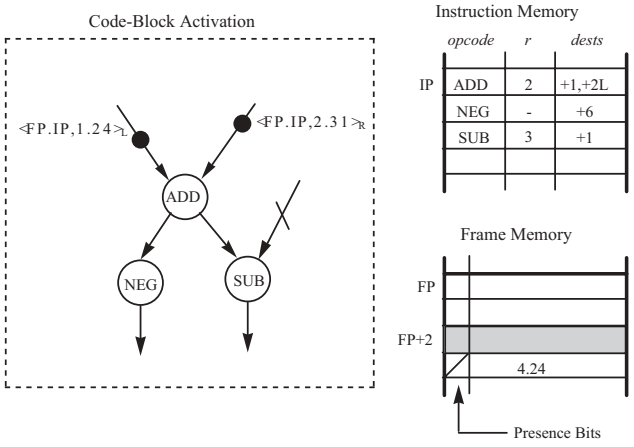

Figure 6. Explicit-token-store representation of a dataflow program.

checking the disposition of the slot in the frame memory pointed to by *FP* + *r*. If the slot is empty, the system writes the token's value in the slot and sets its presence bit to indicate that the slot is full. If the slot is already full, the system extracts the value, leaving the slot empty, executes the corresponding instruction, and communicates the result tokens to the destination instructions by updating the IP according to the destinations encoded in the instruction.

Table 1 lists early dataflow architectures that have been advanced in the literature. Because of the space constraints, additional discussion about these architectures are beyond the scope of this article and the interested reader is referred to the cited references.

Dataflow Architectures of the 1980s and the 1990s

Relying on the lessons learned from early designs, several dataflow prototypes were designed during the 1980s and the 1990s. These include Monsoon, Epsilon-2, EM-4, P-RISC, and TAM. Table 2 summarizes the architectural characteristics of these designs (18–23).

These prototypes use the dynamic dataflow paradigm, primarily because of the success of direct matching of tokens proposed in ETS. The concept of a code-block in ETS permitted localization and efficient management of tokens. Activation frames can be allocated for different iterations of a loop, thus permitting the "unfolding" of loops.

Another major architectural change was the integration of the control-flow sequencing with the dataflow model. Dataflow architectures that are based on the pure dataflow model, such as the Manchester dataflow machine, provide well-integrated synchronization at the instruction level. However, this process is very inefficient when compared with the synchronization used in control-flow systems. It has been shown that it is more efficient to assign some of these responsibilities to the compiler and to use a simpler control-flow sequencing at run-time. The overhead of constructing and communicating result tokens can be reduced by using processor registers to hold intermediate results (similar to control-flow processors). The hybrid of dataflow flow with control-flow sequencing and usage of registers can be found in EM-4, and the Epsilon-2.

In contrast to these hybrid systems, the threaded abstract machine (TAM) provided a conceptually different perspective on the implementation of the dataflow model of computation. In TAM, the execution model for fine-grain parallelism is supported by an appropriate compilation strategy and program representation rather than by elaborate hardware. By assigning the synchronization, the scheduling, and the storage management tasks to the compiler, the use of processor resources can be optimized for the expected case, rather than for the worst case. In addition, because the scheduling of threads is visible to the compiler, TAM allowed for a more flexible use of registers across thread boundaries.

Recent Dataflow Projects

Since the mid 1980s, computer architecture has expended a considerable effort in exploitation of ILP as a means to improve performance. These efforts manifested in the

Table 2. Dataflow architectures of 80's and 90's.

Architecture	Key features
Monsoon (18)	• Joint venture between MIT and Motorola was an outgrowth of the MIT Tagged-Token Dataflow Architecture. • A collection of processing elements communicating with each other and a set of interleaved I-structure memory modules through a multistage packet switching network. • Direct matching of tokens based on the Explicit Token Store concept.
EM-4 (19)	• A highly parallel dataflow multiprocessor based on the SIGMA-1 project. It was an attempt to simplify the architecture by a RISC-based single-chip design, a direct matching scheme, and use of strongly connected arc model. • Use of registers to reduce the instruction cycle time and the communication overhead of transferring tokens. • Integration of a token-based circular pipeline and a register-based advanced control pipeline. • The prototyped processing element consists of a memory module and a single chip processor called EMC-R.
Epsilon-2 (20)	• Epsilon-2 is a multiprocessor dynamic dataflow model evolved from the Epsilon-1 project. It is composed of a set of processing modules contented via a global interconnection network. • Direct matching of tokens. • Repeat fan-out mechanism to reduce the overhead in copying tokens. • Control-flow type of sequencing and use of registers. Register contents are not necessarily preserved across grain boundaries. • Load balancing (adaptive routing).
P-RISC (21)	• P-RISC is a multiprocessor architecture strongly influenced by Iannucci's dataflow/von Neumann hybrid architecture. It utilizes a RISC-like Instruction set and its generalization for parallel-RISC. Can use both conventional and dataflow compiling technologies. • Application of multithread using a token queue and circulating thread descriptors. • Introduction of Fork and Join instructions to spawn and synchronize multiple threads. • Synchronization of memory accesses by using I-structure semantics.
TAM (22)	• Placing all synchronization, scheduling, and storage management responsibility for execution of fine-grain parallelism explicit and under compiler control to relieve hardware complexity. This allows execution of dataflow languages on conventional control-flow processors. • Providing a basis for scheduling a number of threads within an activation as a quantum while carrying values in registers across threads. • Once an activation is made resident, all enabled threads within the activation execute to completion. • Having the compiler produce specialized message handlers as inlets to each code-block. • A prototype TAM instruction set, TL0 (Threaded Language Version 0), has been developed at the University of California at Berkeley.
*T (23)	• Tokens do not carry data, only continuations. • Provides limited token matching. • Overhead is reduced by off-loading the burden of message handling and synchronization to separate coprocessors.

design and the implementation of the so called superscalar, Very Long Instruction Word, super-speculative, and super-pipelined organizations. In these organizations, ILP is exploited through deep pipelining and out-of-order execution of instructions. Aggressive exploitation of ILP is made possible by wide dispatch and issue of instruction, by a large issue buffers, by a large number of physical registers for register renaming, by a large number of functional units, and by a speculative execution of branches. Whether the instruction placement and issue are done statically as in VLIW architecture, or dynamically as in superscalar paradigm, the hardware complexity of these architectures, combined with the diminishing performance gains, renewed an interest in dataflow processing. This interest has resulted in several dataflow based architectures, such as the scheduled dataflow (SDF) (24), the EDGE (Explicit Data Graph Execution) (25), the WaveScalar (26), and the D^2NOW (Data-Driven Network of Workstations) (27).

Scheduled Dataflow (SDF). Unlike instruction level dataflow systems, SDF (24) uses dataflow-like synchronization at the thread-level, and control-flow semantics within a thread. A thread is allocated as an activation frame for receiving its inputs, similar to Cilk (28). A thread is enabled for execution when it has received all its inputs, and completes execution without interruption (viz., non-blocking threads). This approach minimizes instruction level communication, and because SDF threads are very fine-grained (typically a basic block), the amount of parallelism lost because of the sequential execution of instructions within a thread is minimal. Additionally, SDF decouples completely all memory accesses from execution pipelines, resulting in overlapped execution of threads. When a thread is enabled, SDF allocates a register set for the thread. Data is *pre-loaded* into the register set context prior to its scheduling on the execution pipeline. After a thread completes execution, the results are *post-stored* from its registers into memory. All memory accesses are performed by synchronization processors (SPs). The execution engines (EPs) rely on in-order execution of instructions within a thread. This architecture exploits two levels of parallelism: Multiple threads can be active simultaneously, permitting thread level parallelism, and the three phases of a thread execution (pre-load, execute, and post-store) can be overlapped with those of other threads. It is also possible to select appropriate number of SPs and EPs to meet application needs. Thread level speculation to improve performance of imperative programs is simplified in SDF system. Similar to the WaveScalar design, epoch numbers are associated with threads along with extended cache coherency protocols to commit (post-store) the results of a speculative thread in program order.

Explicit Data Graph Execution (EDGE). EDGE (25) is a static placement dynamic issue instruction model. It is designed to allow direct instruction communication: hardware delivers a producer's output directly as an input to a consumer instruction, (i.e., fine grained instruction scheduling). The TRIP architecture is an instantiation of an EDGE design. The TRIP prototype is a collection of 16 execution units that communicate with each other via a thin operand routing network. Each processing element includes an integer unit, a floating point unit, an operand router, and an instruction buffer of depth 128 (to hold multiple instructions and their operands). The scheduler determines which instructions to be assigned in each processor buffer (viz., Static placement). However, the availability of operands determines the order of the execution (viz., Dynamic issue).

WaveScalar. Similar to EDGE, WaveScalar (26) is a tiled architecture. It is a tagged-token dynamic dataflow machine composed of processing elements. Instructions are bound dynamically to processing elements during the execution phase. It should be noted that once an instruction is bound to a processing element, it can remain there for many dynamic executions. A processing element is composed of an interface to receive data tokens: a storage medium to store data tokens awaiting their matching partner. Upon the execution of an instruction, the output tokens are routed to the consumer processing element(s). To reduce the communication costs, the processing elements are connected through a hierarchical interconnection infrastructure—pairs of processing elements are coupled into *pods* sharing ALU results via a common bypass network. Four pods are grouped into *domains* that communicate over a set of pipelined busses. Four domains form a cluster supported by conventional memory hierarchy. To build larger machines, multiple clusters can be connected with each other by a grid-based, on-chip network.

Data-Driven Network of Workstations. D^2NOW (27) is a collection of off-the-shelf Pantium microprocessors interconnected through a fine-grained interconnection network that supports the thread level synchronization. The design is based on an earlier Decoupled Data Driven model of execution and in principal similar to the SDF model (24). To tolerate the communication latency, D^2NOW employs three mechanisms: fine-grained communication, medium-grained communication, and coarse-grained communication. The fine-grained communication is used for consumer identification and for data-token transfer. The medium-grained communication is for the medium size messages that identify a code block. The coarse-grained communication is through the Ethernet to support large data block transfers.

SOME RESEARCH ISSUES

Handling Data Structures

In pure dataflow model, no concept of a variable exists and data is exchanged in the form of tokens flowing between instructions (or if memory is used to store data, then variables can only be assigned a value once – the single assignment principle). To apply this property to arrays and structures, the entire structure must be carried as tokens (or new arrays and structures must be allocated, by copying unchanged items and by assigning new values to the elements that have been modified). In practical dataflow systems, a more efficient treatment of structures and of

arrays is needed. The proposed solutions can be classified as either direct access or indirect access methods.

Direct Access Method. The direct access scheme treats each array elements as individual (scalar) data tokens — which eliminates the concept of an array (or structure). The token relabeling scheme proposed by Gaudiot is an example of direct access method (29). In this approach, tokens are identified by tags, associating the values with specific array elements. Although this method is simple, it requires entire data structures be passed from one node to the next or to be duplicated among different nodes. Moreover, in many applications, the notion of array as a single entity cannot be done away with completely, and thus the direct access method is inappropriate for such applications.

Indirect Access Method. In an indirect access scheme, arrays are stored in special (separate) memory units and their elements are accessed through explicit "read" and "write" operations. For example, in MIT, static dataflow machine arrays are represented as a heap forming a tree (30). VAL (see section on Dataflow Languages) provides constructs to generate and to access arrays. Arrays can be appended with new values, and arrays (and elements of arrays) are accessed using pointers. Modified elements can be made inaccessible by using reference counts with pointers (when the count becomes zero the element becomes inaccessible). The disadvantages of this method are: O(log n) time to access successive elements of an array and sequential nature of the append operations (only one element of an array can be modified at a time), which limits the performance of the system. It also becomes necessary to perform garbage collection of inaccessible elements.

I-structure. I-structures are asynchronous array-like structures that include a "presence" bit with each element of the structure, thus preventing access to undefined array elements and enforcing single assignment property (31).

University of Manchester Approach. This approach combines the concept of streams (i.e., a sequence of values communicated between two portions of a code) with conventional arrays (32). However, in contrast to streams, the size of the array structure must be known at the time it is created. Thus, a finite component is defined as a collection of elements, a "unit," on which the basic storage operations are performed. This scheme implies that the modification of any element(s) in an array requires copying the entire array. Sisal language (see the Dataflow Languages section) provides constructs to create and access arrays, as well as streams. An enhanced version permitted "in-place" updates to alleviate the need to copy unaffected elements.

Hybrid Scheme. The basic idea behind the hybrid scheme is to associate a template, called the structure template, with each conceptual array (33). For selective updates, this minimizes copying by allowing only the modified elements to be appended to the new array. Each array is represented by a hybrid structure that consists of a structure template and a vector of array elements. A structure template is subdivided into three fields:

- The reference count field; an integer indicating the number of references to the array,
- The location field; a string of 1's and 0's where the length of the string equals the total number of elements in the array. Each location bit determines whether the desired array element resides in the vector indicated by either the left ("0") or the right ("1") pointer, and
- The status bit (S); when an array is initially created, the status bit (S) is initialized to "0," which indicates that the vector contains the original array. Whenever a modification is made to an array with more than one reference, a new hybrid structure is created (the status bit set to "1") where all the modified elements can be accessed from the vector pointed by the right pointer. The sharing of array elements between the original and the modified array is achieved by linking the left pointer of the modified hybrid structure back to the original hybrid structure.

Program Allocation

To achieve maximum parallelism, programs must be partitioned and assigned to available resources. The goal is to maximize the parallelism (partition program into independent executable units) while minimizing communication among the executable units (by assigning dependent units to the same processing element). It has been shown that obtaining an optimal allocation of a graph with precedence requirements is an NP-complete problem. Two main (heuristic) approaches exist to allocate subtasks of a dataflow graph: static and dynamic. In static allocation, the tasks are allocated at compile-time using global information about the program and system resources. A dynamic allocation uses run-time information on processing loads and on program behavior to distribute tasks.

A number of heuristic algorithms have been developed for the allocation problem based on critical path list schedules. The basic idea behind these approaches is to assign a weight to each node of a directed graph that equals the maximum execution time from that node to an exit node (i.e., critical path). An ordered list of nodes is constructed according to their weights, which is then used to assign dynamically nodes with highest weights to processors as they become available. One major problem with critical path list schedules is the communication among the nodes. Enforcing only critical path scheduling, without considering the communication overhead, will not necessarily minimize the overall execution time.

In response, Ravi et. al. (34) proposed a variation of the critical path list scheduling which takes into account inter processor communication. In this method, rather than simply choosing the topmost node on the list, several top candidates whose critical paths fall within a certain range are considered for allocation. From this set of candidates, a node is selected which maximizes savings in communication time. To determine the compromise between computation and communication costs, the vertically layered (VL) allocation scheme was proposed in Ref. (35). The VL allocation scheme consists of two phases: separation and

optimization. The basic idea behind the separation phase is to partition a dataflow graph into vertical layers such that each vertical layer represents a set of data dependent nodes that are executed in sequence (i.e., a thread). A density factor is used to distribute the directed paths among the processors. The optimization phase attempts to minimize the inter-processor communication costs by considering whether the inter-processor communication overhead offsets the advantage gained by overlapping the execution of two subsets of nodes on separate processors (collapsing the vertical layers assigned to different processors). The VL allocation scheme succeeds in balancing the load among the processors, however, by its very nature, it may not always reduce the total execution time.

A more general approach to program allocation was proposed by Sarkar and Hennessy (36). In contrast to the VL allocation scheme, this approach uses a greedy approximation algorithm. The algorithm begins with the trivial partition that places each node in a separate block. A table that represents the decrease in the critical path length obtained from merging a pair of blocks is maintained. It then merges iteratively blocks that result in the largest decrease in the critical path length. The algorithm is terminated when no remaining merger could possibly reduce the critical path length.

Despite the effectiveness of the aforementioned allocation schemes, one major problem still remains unresolved — the issue of handling dynamic parallelism. For example, a dynamic architecture unfolds a loop at run-time by generating multiple instances of the loop body and attempts to execute the instances concurrently. However, a single processor does not allow two simultaneous executions of a node, consequently, mapping the source dataflow graphs to processors, without special provisions to detect dynamic loop unfolding, results in the inability to exploit parallelism fully.

One solution is to provide a code-copying facility, where an instruction within a code block is duplicated among the available resources. Arvind has proposed a mapping scheme in which the instructions within a code block (called the logical domain) are mapped onto available procesors (called the physical domain) based on a hashing scheme (37). For example, if a physical domain consists of n processors, then the destination processor number can be $\text{processor}_{\text{base}} + i \bmod n$, where i is the iteration number. This will distribute the code uniformly over the physical domain. Because each of the n processors has a copy of the code, n iterations may be executed simultaneously. However, because not all program constructs can be unfolded in this manner, the question still remains as to how dynamic parallelism can be detected at compile-time effectively.

Cache in Dataflow

Multithreading can address memory latencies by context-switching to other thread while awaiting a memory access. In pure dataflow, each instruction can be viewed as a thread, causing excessive overheads. ETS based models and hybrid systems (see the Dataflow Architectures of the 1980s and the 1990s section) have created coarser grained threads using code blocks. Within the context of dataflow, threads are nonblocking: A thread is enabled for execution when it receives all inputs, and executes to completion without interruption or context switch. The nonblocking makes it difficult to overcome memory latencies because a thread cannot be context switched during its execution. Cache memories provide a solution, but in pure dataflow, because instruction execution is based on the availability of data, localities of instructions and data cannot be determined easily, making the inclusion of cache memories wasteful. Several innovative proposals for synthesizing localities in the context of dataflow exist. In Ref. (38), the concept of simultaneity of execution is used to define localities with code. A weight that represents the distance from the root is assigned to each dataflow node. The nodes with the same weight are then clustered on the same (cache) page. This strategy partitions the dataflow graph into K horizontal layers, such that the nodes in layer K_i are data independent from each other (hence they can likely be executed in parallel) and are data dependent on nodes in layer K_{i-1} $(1 < \text{i} \leq \text{K})$.

Other approaches to improve localities and cache in dataflow can be found in Ref. (39). Partitioning dataflow programs into threads will have a direct impact on localities. Allocation of threads to processing resources should use "cache affinities" to minimize cache misses and conflicts. An important issue in multithreading is the partitioning of programs into multiple sequential threads (see the Program Allocation section). The costs associated with creating threads and synchronization among threads will impact the granularity of threads and placement of threads. Schauser et al. (40) proposed a partitioning scheme using dual graphs. A dual graph is a directed graph with data, control, and dependence arcs: A data arc represents the data dependence between producer and consumer nodes. A control arc represents the scheduling order between two nodes, and a dependence arc specifies long latency operation caused by the message handlers (i.e., inlets and outlets) sending/receiving messages across code-block boundaries. The actual partitioning is performed using only the control and the dependence edges by first grouping the nodes based on dependence sets. The dependence sets are used to create safe partitions with no cyclic dependencies. A safe partition has the following characteristics: (1) no output of the partition needs to be produced before all inputs to the partition are available, (2) when the inputs to the partition are available, all the nodes in the partition can be executed, and (3) no arc connects a node in the partition to an input node of the same partition. A number of optimization techniques are performed on initial partitions to reduce synchronization costs. The output of the partitioner is a set of threads where the nodes in each thread are executed sequentially and the synchronization requirement determined statically only occurs at the beginning of a thread. SDF and TAM (see the Dataflow Architectures section) use similar ideas for thread generation.

In the static dataflow architecture, localities can be exploited by concentrating on the static order of the dataflow program. The dynamic approach permits the activation of several instances of a node during run-time. To exploit the temporal and the spatial localities in dataflow programs that run on dynamic dataflow models, it is

necessary to separate instruction memory from the operand memory. However, asynchrony of the dataflow instructions means frequent context switching, and in general, lack of temporal and spatial localities in accessing instruction and operand memories (41). To cope with these problems, one needs to adopt proper mechanisms to partition the dataflow graphs into subgraphs, to allocate subgraphs among processors, and to control the number of instances of a subgraph in a processor. It should be noted that because of the functional and asynchronous nature of the dataflow instructions, the addresses of the nodes in a dataflow graph can be set as desired without affecting the result of the execution. This property should be the basis of establishing localities in dataflow programs. Moreover, in support of the cache organization, one should study the effectiveness of the traditional statistical replacement algorithms (e.g., LRU) for instruction and operand memories. Therefore, in a processor with the load control mechanism, a sophisticated deterministic algorithm to replace dataflow blocks needs to be developed. Finally, the operand memory plays a dominant role to achieve satisfactory performance in a dataflow machine, and hence the operand cache must be managed effectively. In a dataflow machine, it is not only necessary to maintain spatial locality for the input arguments of a code-block (frame), but also is necessary to maintain spatial locality for the result tokens of the code-block. In the other words, the cache management must keep track of several active frames to avoid cache misses in accessing arguments while storing the results. These design principles motivated the organizations of operand and instruction caches in the literature (41).

CONCLUSION

As modern architects find it difficult to design highly parallel architectures that can exploit high degrees of instruction level parallelism, it may be time to look back to dataflow model of computation. The dataflow model was investigated in 1970s and 1980s but no commercially viable systems were implemented. Nevertheless, several features of the dataflow principle and dataflow computation have found their place in modern processor architectures and compiler technology. Most modern processors use complex hardware techniques to detect data hazards, control hazards, and dynamic parallelism — to bring the execution engine closer to an idealized dataflow engine. Some researchers have proposed hybrid designs in which the dataflow scheduling is applied only at thread level (i.e., macro-dataflow), whereas each thread is comprosed of conventional control-flow instructions.

The advances from the development of dataflow projects indicate potential high performance computation based on the dataflow principles. However, before a successful implementation of dataflow machines is possible, the various issues discussed in this article must be resolved.

It is our contention that a more careful mix of dataflow principles with recent technological advances will pave the way to future tera and peta instructions per second performance. Current multicore and multithreaded systems do not scale well. Instruction level dataflow implementations potentially can scale with processing resources, but they require excessive hardware support for synchronization, distribution, and communication among the instructions. A combination of static (compile time) and dynamic techniques for the creation and distribution of threads (or a unit of concurrent activity) may provide a balance between performance and hardware complexity.

The implementation of imperative memory systems within the context of a dataflow model is yet another issue that is not addressed satisfactorily. In addition to the management of structures, techniques for the management of pointers, dealing with aliasing, and dynamic memory management are needed. Ordering of memory updates (a critical concept in shared memory concurrency) is an alien concept to pure dataflow. However, to be commercially viable, it is essential to provide shared memory based synchronization among concurrent activities. Some ideas such as those presented in Wavescalar and SDF hold some promise in this connection.

We believe that several factors are motivating a renewed interest in the design and implementation of scalable dataflow processors. These include the recent technological advances in increased chip density; the complex interconnections among multiple processing elements on a single chip (Network-on-a-chip); the hardware complexity of the superscalar, super pipeline, and VLIW architectures and the diminishing performance gains of these systems with additional hardware; the large and multi-level caches; the compliers that perform extensive global and interprocedural analyses to extract as much parallelism as possible, and finally, the success of the recent dataflow projects (Recent Dataflow Projects section). At the same time, if dataflow architecture is to address the challenges of future processing architectures containing hundreds, if not thousands, of processing elements, it is necessary to evaluate carefully different forms of dataflow organizations for their suitability for implementation.

BIBLIOGRAPHY

1. W. B. Ackerman, Dataflow languages, *IEEE Computer*, **15** (2): 15–23, 1982.
2. R. S. Nikhil, *Id World Reference Manual*, MIT Laboratory for Computer Science, Cambridge, MA, 1985.
3. J. Feo, D. Cann, and R. Oldehoeft, A report on the Sisal language project, *J. Parallel Distribut. Comput.*, **10**: 249–365, 1990.
4. R. Oldehoeft and D. Cann, Applicative parallelism on a shared-memory multiprocessor, *IEEE Software*, **5** (1): 62–70, 1988.
5. Sisal Lives, Available: http://sisal.sourceforge.net/
6. B. Lee and A. R. Hurson, Dataflow architectures and multithreading, *IEEE Computers*, **27** (8): 27–38, 1994.
7. J. B. Dennis and D. P. Misunas, A preliminary architecture for a basic dataflow processor, *Proc. Symposium on Computer Architecture*, 1975, pp. 126–132.
8. J. R. Gurd, C. C. Kirkham, and I. Watson, The manchester prototype data-flow computer, *Comm. ACM*, **28** (1): 34–52, 1985.
9. Arvind and D. E. Culler, Dataflow Architectures, *Ann. Rev. Comp. Sci.*, **1**: 225–253, 1986.

10. A. L. Davis, The architecture and system Method of DDM1: A recursively structured data driven machine, *Proc. Symposium on Computer Architecture*, 1978, pp. 210–215.
11. M. Cornish, The TI dataflow architecture: The power of concurrency for avionics, *Proceedings of Third Conference on Digital Avionics Systems*, 1979, pp. 19–25.
12. Y. M. Chong, Dataflow chip optimizes image processing, *Computer Design*, 97–103, 1984.
13. N. Takahashi and M. Amamiya, A dataflow processor array system: Design and analysis, *Proc. Symposium on Computer Architecture*, 1983, pp. 243–250.
14. M. Kishi, H. Yasuhara, and Y. Kawamura, DDDP: A distributed data driven processor, *Proc. Symposium on Computer Architecture*, 1983, pp. 236–242.
15. D. F. Snelling, The design and analysis of a Stateless Data-Flow Architecture, *Tech. Report UMCS-93-7-2*, University of Manchester, 1993.
16. T. Shimada et al., Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations, *Proc. Int. Symposium on Computer Architecture*, 1986, pp. 226–234.
17. N. Ito, M. Kishi, E. Kuno, and K. Rokusawa, The data-flow based parallel inference machine to support two basic languages in KL1, *Proc. IFIP TC-10 Working Conf. Fifth Generation Comp. Arch.*, 1985, pp. 123–145.
18. D. E. Culler and G. M. Papadopoulos, The explicit token store, *J. Parall. Distribut. Comput.*, **10**: 289–308, 1990.
19. M. Sato et al., Thread-based programming for EM-4 hybrid dataflow machine, *Proc. Symposium on Computer Architecture*, 1992, pp. 146–155.
20. V. G. Grafe and J. E. Hoch, The epsilon-2 multiprocessor system, *J. Paral. & Distribut. Comput.*, **10**: 309–318, 1990.
21. R. S. Nikhil and Arvind , Can Dataflow Subsume von Neumann Computing?*Proc. Int. Symposium on Computer Architecture*, 1989, pp. 262–272.
22. D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. Eicken, TAM — A compiler-controlled threaded abstract machine, *J. Paral. Distribut. Comput.*, **18**: 347–370, 1993.
23. R. S. Nikhil, G. M. Papadopoulos, and Arvind , *T: A Multithreaded Massively Parallel Architecture, *Proc. Int. Symposium on Computer Architecture*, 1992, pp. 156–167.
24. K. M. Kavi, R. Giorgi, and J. Arul, Scheduled dataflow: execution paradigm, architecture, and performance evaluation, *IEEE Trans. Comp.*, **50** (8): 834–846, 2001.
25. D. Burger et al., Scaling to the end of silicon with EDGE architectures, *IEEE Computer*, **37** (7): 44–55, 2004.
26. S. Swanson et al., Area-performance trade-offs in tiled dataflow architectures, *IEEE*, **34** (2): 314–326.
27. C. Kyriacou, P. Evipidou, and P. Trancoso, Data-driven multithreading using conventional microprocessors, *IEEE Trans. Parallel Distribut. Sys.*, **17** (10): 1176–1188, 2006.
28. R. D. Blumofe et al., Cilk: An Efficient Multithreaded Run-time System, *ACM Symposium on Principles and Practice of Parallel Programming (PPoP)*, 1995.
29. J.-L. Gaudiot and Y. H. Wei, Token relabeling in a tagged token data-flow architecture, *IEEE Trans. Comp.*, **38** (9):1225–1239, 1989.
30. W. B. Ackerman, A structure processing facility for dataflow computers, *Proc. of the International Conference on Parallel Processing*, 1978, 166–172.
31. Arvind, R. S. Nikhil, and K. K. Pingali, I-structures: Data structures for parallel computing, *Proc. of the Workshop on Graph Reduction*, Los Alamos, NM, 1986.
32. L. M. Patnaik, R. Govindarajan, and N. S. Ramadoss, Design and performance evaluation of EXMAN: an extended manchester dataflow computer, *IEEE Trans. Comp.*, **35** (3): 229–243, 1986.
33. B. Lee, A. R. Hurson, and B. Shirazi, A hybrid scheme for processing data structures in a dataflow environment, *IEEE Trans. Parallel Distribut. Sys.*, **3** (1): 83–96, 1992.
34. T. M. Ravi, M. D. Ercegovac, T. Lang, and R. R. Muntz, Static allocation for a dataflow multiprocessor system, *2nd Int. Conference on Supercomputing*, 1987.
35. B. Lee, A. R. Hurson, and T. Y. Feng, A vertically layered allocation scheme for dataflow computers, *J. Parallel Distribut. Comput.*, **11**: 175–187, 1991.
36. V. Sarkar and J. Hennessy, Compile-time partitioning and scheduling of parallel programs, *Proc. SIGPLAN Symposium on Compiler Construction*, 1986, pp. 17–26.
37. Arvind , Decomposing a program for multiprocessor system, *Proc. of the International Conference on Parallel Processing*, 1980, pp. 7–14.
38. J. T. Lim, A. R. Hurson, and L. D. Pritchett, searching for locality in program graphs, *Paral. Comput. Technol.*, LNCS **2763**: 276–290, 2003.
39. A. R. Hurson, K. Kavi, B. Lee, and B. Shirazi, Cache memories in dataflow architectures: a survey, *IEEE Parall. Distribut. Technol.*, **4** (4): 50–64, 1996.
40. K. E. Schauseret al., Compiler-controlled multithreading for lenient parallel languages, *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1991, pp. 50–72.
41. M. Takesau, Cache memories for dataflow machines, *IEEE Trans. Comput.*, **41** (6): 677–687, 1992.

FURTHER READING

G. M. Papadopoulos, *Implementation of a General-Purpose Dataflow Multiprocessor*, Cambridge, MA: *MIT Press*, 1991.

K. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, Design of cache memories for multithreaded dataflow architecture, *Proc. of Symposium on Computer Architecture*, 1995, 253–264.

ALI R. HURSON
University of Missouri-Rolla
Rolla, Missouri
KRISHNA M. KAVI
The University of North Texas
Denton, Texas