Concurrency, Synchronization, Speculation - the Dataflow Way

Krishna Kavi, Domenico Pace and Charles Shelor

November 23, 2013

Abstract

This chapter provides a brief overview of dataflow, including concepts, languages, historical architectures, and recent architectures. It is to serve as an introduction to and summary of the development of the dataflow paradigm during the past 45 years. Dataflow has inherent advantages in concurrency, synchronization, and speculation over control flow or imperative implementations. However, dataflow has its own set of challenges to efficient implementations. This chapter will address the advantages and challenges of dataflow to set a context for the remainder of this issue.

Contents

Acronym	Section	Definition
AQ	5	Acknowledgement Queue of D^2NOW
CU	5	Computing Unit in Codelet
D^2NOW	5	Data-Driven Network of Workstations
DDM	5	Data Driven Multithreading model
DFE	5	DataFlow Engine in Maxeler
D-TSU	5	Distributed Thread Scheduling Unit in Teraflux
DU	5	Data cache Unit in TRIPS
EDGE	5	Explicit Data Graph Execution in TRIPS
EP	5.7	Execution Pipeline of SDF
ETS	4	Explicit Token Store
EU	5	Execution Unit in TRIPS
EXC	5.7	Execution Continuation of SDF
FP	4	Frame Pointer
FSS	2.1	Firing Semantic Sets
GM	5	Graph Memory of D^2NOW
GU	5	Global control Unit in TRIPS
IP	4	Instruction Pointer
IU	5	Instruction cache Unit in TRIPS
L-TSU	5	Local Thread Scheduling Unit in Teraflux
NIU	5	Network Interface Unit of D^2NOW
PE	5	Processing Element of WaveScalar
PLC	5.7	Pre-Load Continuation of SDF
PSC	5.7	Post-Store Continuation of SDF
RQ	5	Ready Queue of D^2NOW
RU	5	Register Unit in TRIPS
SDF	5.7	Scheduled DataFlow
SISAL	2.1	Streams and Iteration in a Single Assignment
SM	5	Synchronization Memory of D^2NOW
SP	5.7	Synchronization Pipeline of SDF
SU	5	Scheduling Unit in Codelet
SU	5.7	Scheduling Unit of SDF
TLS	5.7	Thread Level Speculation
TP	5	Threaded Procedure in Codelet
TRIPS	5	Tera-op Reliable Intelligently adaptive Processing System
TSU	5.7	Thread Scheduling Unit of SDF
TSU	5	Thread Synchronization Unit of D^2NOW
VAL	2.1	Value based programming language
WTC	5.7	Waiting Continuation of SDF

Table 1: Table of Abbreviations

1 Introduction

Achieving high performance is possible when multiple activities (or multiple instructions) can be executed concurrently. The concurrency must not incur large overheads if it is to be effective. A second issue that must be addressed while executing concurrent activities is synchronization and/or coordination of the activities. These actions often lead to sequentialization of parallel activities, thus defeating the potential performance gains of concurrent execution. Thus, effective use of synchronization and coordination are essential to achieving high performance. One way to achieve this goal is through speculative execution, whereby it is speculated that concurrent activities do not need synchronization or coordination; or predict the nature of the coordination. Successful speculation will reduce sequential portions of parallel programs; but mis-prediction may add to execution times and power consumption since the speculatively executed activities must be undone and the activity must be restarted with correct synchronization.

The dataflow model of computation presents a natural choice for achieving concurrency, synchronization and speculations. In the basic form, activities in a dataflow model are enabled when they receive all the necessary inputs; no other triggers are needed. Thus all enabled activities can be executed concurrently if functional units are available. And the only synchronization among computations is the flow of data. In a broader sense, coordination or predicate results can be viewed as data that enables or coordinates dataflow activities. The functional nature of dataflow (eliminating side-effects that plague imperative programming models) makes it easier to use speculation or greedy execution; unnecessary or unwanted computations can be simply discarded without any need for complex recovery on mis-speculations.

In this chapter we will introduce well-understood dataflow models of computation in section 2. We will review programming languages that adhere to dataflow principles in section 3. We will present historical attempts at developing architectures implementing the dataflow models along with a discussion of the limitations of dataflow encountered by these architectures in section 4. We will present some recent variations of the dataflow paradigm that could potentially lead to efficient architectures in section 5. Finally we will include our conclusions and prognosis on the future of the model as a viable alternative to control-flow systems in section 6.

2 Dataflow Concepts

2.1 Dataflow Formalism

The dataflow model of computation is based on the use of graphs to represent computations and flow of information among the computations. The signal processing community uses a visual dataflow model to specify computations. Some examples of environments used by this community include Signal [8], [34], Lustre [21], Ptolemy [10]. In this chapter our focus is on general-purpose programming and general-purpose computer architectures. Hence we will not review programming models, languages or environments for specific or restricted domains, however Lee [37] provides a good survey on such languages and models.

One of the earliest dataflow models is called Khan [25] process networks. A program in this model consists of channels, processes, and links connecting these processes. Figure 1 shows an example program and the corresponding graphical representation of the process network. In principle channels can be viewed as arrays or lists (sequence of values). It is also possible to associate firing semantics with a process to define necessary input sequences and non-determinism. We again refer the reader to the survey by Lee [37]. A process takes the required sequences of inputs from its input channels and creates sequences on

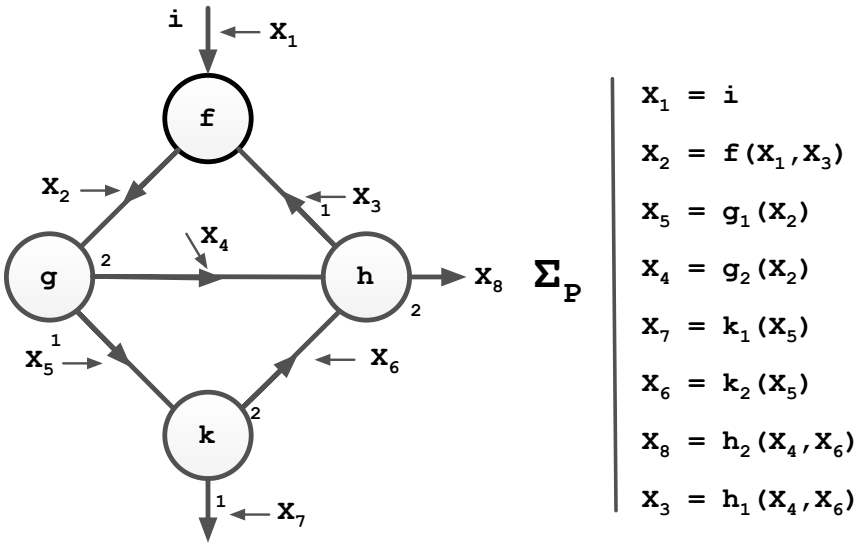


Figure 1: Khan Process Networks Example

output channels. To facilitate continuous execution and concurrency, one can view the sequences of data on channels as partially ordered. A set of inputs may contain multiple, partially ordered inputs, where the lack of ordering among subsequences allows for concurrency.

There have been several formalisms that have extended Khan process networks. We will use specific notations to describe a process network, which resemble the most common view of dataflow graphs to graphically represent computations. We will use the notations from [27] in the rest of this section.

A dataflow graph is a directed graph in which the elements are called links and actors [14]. In this model actors (nodes) describe operations while links receive data from a single actor and transmit values to one or more actors by way of arcs; arcs can be considered as channels of communication, while links can be viewed as placeholders or buffers.

G = < A ∪ L, E>

where A = {*a1, a2, ..., an*} is the set of actors; L = {*l1, l2, ..., lm*} is the set of links and

E ⊆ (A X L) ∪ (L x A) is the set of edges connecting links and actors.

In its basic form, activities (or nodes) are enabled for execution when all input arcs contain tokens and no output arcs contain tokens. However, if we consider arcs as (potentially infinite) buffers, as in Kahn's process networks, the condition related to output arcs can be removed. The values in these buffers can be viewed as partially ordered sequences allowing for multiple invocations of activities. This concept is the basis of tagged-token dataflow architectures.

The model described above, without semantic interpretation of activities has been shown to be isomorphic to free choice Petri nets [28]. In addition, it was used as a model for concurrent processes with semantics of actors defined using predicate logic and using partial ordering of values at links [16], [29]. The model was then used to derive conditions to assure liveness and safety properties of concurrent processes. By attaching semantics with nodes (or activities), or describing the computation performed by the nodes

using some programming notation, dataflow graphs can be used for parallel programming. A hardware schematic can also be viewed as a dataflow graph where the schematic components are dataflow nodes and the signals on the schematic are the dataflow arcs bringing the outputs of generating nodes to the inputs of receiving nodes.

To be useful as a general purpose programming model, we need to permit conditional operations (such as if..then..else constructs). Thus we can view actors, graphically represented using nodes, as belonging to one of three types: activities, predicates, gates (or conditional execution). Activities represent computations that consume data on input arcs and produce data on output arcs. Predicates consume input data and produce Boolean (true or false) outputs. Gates require two types of inputs: a Boolean input and data inputs. Gates use the value of the Boolean input to decide how to process inputs or generate outputs. Control arcs (which enter or leave control links) have tokens of type Boolean (true or false). Data arcs (which enter or leave data links) have tokens of any data type (e.g., integer, real, or character). In section 3.1 we will introduce how complex data structures such as arrays and structures can be handled in dataflow languages and architectures.

Generalized Firing Semantic Sets (FSS). The basic firing rule adopted by most dataflow researchers requires that all input arcs contain tokens and that no tokens be present on the output arcs. This provides an adequate sequencing control mechanism when the nodes in dataflow graphs represent primitive operations. However, if the nodes are complex procedures, or dataflow sub-graphs, more generalized firing control for both input and output links is required. The rules can indicate which subset of input links of an actor must contain values and which subset of output links of the actor must not contain any values for enabling the actor for execution. The ability to enable actions using only subsets of inputs provides flexible scheduling of activities; for example a node which represents a function or a thread may be allowed to initiate execution with a minimum set of required inputs, while additional input are delivered later (but before they are actually needed).

Graphical notations. For the purpose of this chapter, we will simplify the formalism and notations. We will use the graphical representations shown in Figure 2 for activities, predicates, and gates. Using these notations, we can think of writing dataflow programs. The first program shown in Figure 3 is a single computation that does not use conditional statements.

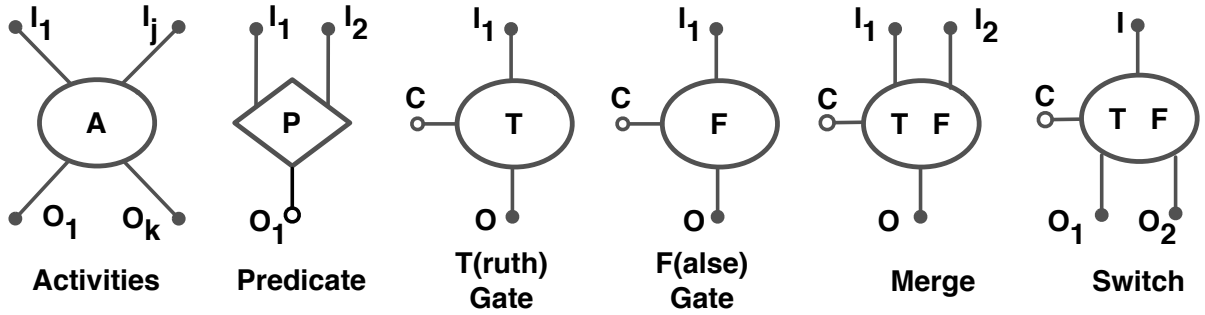


Figure 2: Some graphical notations of Dataflow Actors

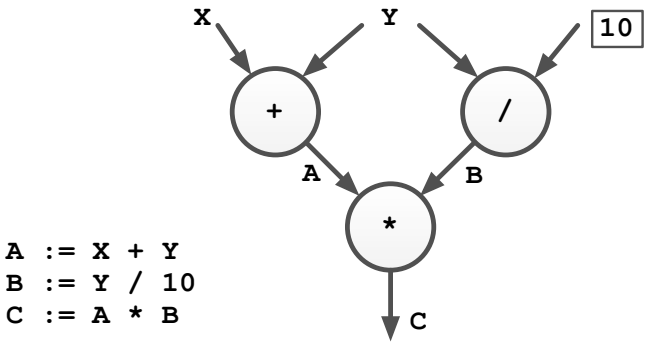


Figure 3: A simple dataflow program example

The second example which is shown in Figure 4 represents how the conditional programming constructs (a loop in this example) can be represented in dataflow. The graph computes the following:

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

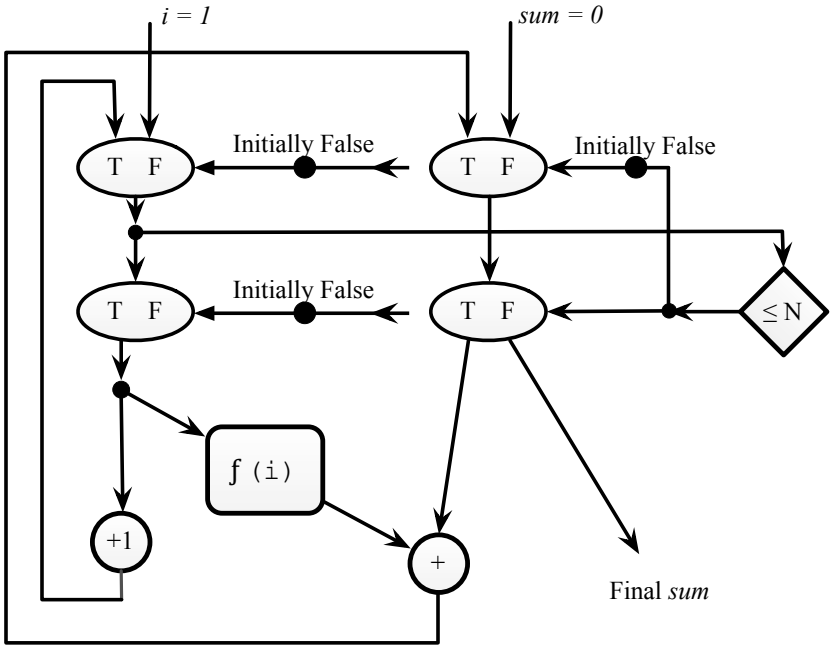


Figure 4: A dataflow program example with a loop and function

As stated previously, an actor in a dataflow graph can actually represent a function or a module so that we can construct more complex programs using simpler programs as sub-graphs.

The dataflow model of computation is neither based on memory structures that require inherent state transitions nor does it depend on history sensitivity. Thus, it eliminates some of the inherent von Neumann

pitfalls described by Backus [7]. We will now describe how the dataflow model is natural for representing concurrency, synchronization, and speculation. Several extensions to basic dataflow have been proposed so that dataflow techniques can be used in a variety of applications, including the use of producer-consumer synchronization, locks and barriers. However, in this section we will rely on the dataflow formalism already introduced here.

Concurrency. An activity in dataflow is enabled when the necessary inputs are available on input arcs; no other trigger is needed. Thus all activities that are enabled can execute concurrently provided sufficient computational units are available. This should be contrasted with von Neumann (or control flow) model, which sequences activities, typically using a program counter. In the simple dataflow program shown in Figure 3 above, both the addition and division operations can be executed concurrently. Likewise, in the second example in Figure 4 the sub-graph representing $f(i)$ can be invoked concurrently for all values of $1 <= \text{i} <= \text{N}$.

In addition to this concurrency, using flexible firing rules mentioned previously, activities represented by sub-graphs (i.e., functions are threads) can be enabled upon receiving a subset of inputs, thus creating an additional level of concurrency, similar to pipelining used in conventional systems.

Synchronization. The only synchronization among activities in dataflow is the data dependency. Thus dataflow only includes true (or Read After Write, RAW) dependency. This is because the dataflow model of computation is neither based on memory structures that require inherent state transitions nor does it depend on history sensitivity: only values have meaning in dataflow and not storage locations. This eliminates side-effects and false dependencies caused by the use of storage that can be modified several times during a program execution. Imperative languages that do use variables for storing data values cause anti (or Write After Read, WAR) and output (Write After Write, WAW) dependencies. These false dependencies force strict order on the execution of activities based on the order in which variables are accessed and modified. The correctness of a program requires that any read access to a variable must return the most recently written value, in the order in which the activities are sequenced (or program order). These false dependencies require complex compile time analyses and/or complex hardware.

Speculation or greedy execution. As stated above, use of variables that can be modified require ordering of statements. In addition, in most modern imperative languages variables may be modified indirectly using pointers. These aspects of imperative programs make it very difficult to determine which activities (or functions, threads) can be executed concurrently, particularly when the dependencies cannot be determined statically. When speculation is used with control flow models, it is necessary to buffer values generated by activities that are enabled speculatively. These values must then be discarded and the effects of the speculative computation must be undone when the speculation fails.

Since the dataflow model does not use variables, but only values, activities can be executed speculatively or aggressively without waiting to completely determine data dependencies (even true dependencies). The output of these speculatively executed activities may be wrong, but they cause no side-effects, and thus can be safely discarded, and an instance of the activity with correct inputs can be initiated. In principle it is possible to complete all possible speculative instances of activities and discard all mis-speculated computations, provided sufficient resources are available. For example, the function f(i) in Figure 4 can be invoked greedily for all values of i, and the results from invocations for $\text{i} > \text{N}$ can be discarded.

3 Dataflow Languages

In this section we will discuss dataflow languages and structures used to implement complex types within dataflow languages. An example is presented to show how dataflow languages can be used for typical problems.

A language that implements the dataflow paradigm must have the following features:

- freedom from side effects;
- locality of effects;
- data dependencies equivalent to scheduling;
- single assignment of named values;
- efficient representation of data structures, recursion, and iteration;

Since scheduling is determined from data dependencies, it is important that the value of variables do not change between their definition and their use. The only way to guarantee this behavior is to disallow the reassignment of variables once their value has been assigned. Therefore, variables in dataflow languages most obey the *single-assignment rule* [13]. This means that they are considered as named values rather than variables. The implication of the single-assignment rule is that the compiler can represent each value as one or more arcs in the resultant dataflow graph, going from the instruction that assigns the value to each instruction that uses that value. An important consequence of the single-assignment rule is that the order of statements in a dataflow language is not important. As long as there are no data loops in the program, the definition of each value, can be placed in any order in the program.

Freedom from side effects is also essential if data dependencies are to determine scheduling. Most languages that avoid side effects disallow global variables and scope rules. In order to ensure the validity of data dependencies, a dataflow program does not permit a function to modify its own input arguments. In addition, since there are no variables or storage locations containing data, the concept of using a pointer to access a storage location is foreign to dataflow. This also eliminates side-effects caused by aliasing. This approach leads to some issues when the program manipulates data structures like arrays.

3.1 Dataflow Structures

One of the key points in the implementation of an efficient dataflow system is that of data structures. As described in the Dataflow Concepts section 2, the dataflow execution model specifies that all data is represented by values, that once created, it cannot be modified. If a node wants to modify a value, it creates a new token, containing new data which is identical to the original data, except for the element that had to be modified. While this data model is perfectly adherent to the theoretical dataflow execution model and for dataflow graphs that deal only with primitive data types, this approach is clearly unsatisfactory for graphs that use more complex data structures, especially in the era of object-oriented programming. The following paragraphs briefly describe some data structure models.

Direct Access method This scheme treats each array element as an individual (scalar) data token, which eliminates the concept of an array or structures. A *tag* could be used to associate the value with a specific array. 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.

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. 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 elements become inaccessible).The main disadvantages of this method are:

- O(log n) to access successive elements of an array;
- append operations are executed sequentially, this results in only one element of an array being modified at a time which limits concurrency;
- a garbage collector is needed to manage elements whose count reaches zero.

Dennis's method. Dennis's method [24] was the first method to provide realistic data structures in a dataflow context by proposing that the tokens in the dataflow program hold not the data itself, but rather a pointer to the data. This schema uses a memory heap in the form of a finite, acyclic, directed graph where each node represents either an elementary value or a structured value which behaves like an indexed array. Each element of a structured value is, in turn, a node that represents either an elementary value or a structured value. The pointers in the data tokens refer to one of this node and a reference count is maintained. A node which is no longer referred to, either directly or indirectly, in the graph is removed by an implicit garbage collector. This method behaves in the following way:

- whenever an elementary value is modified, a new node is simply added to the heap;
- whenever a structured value is modified, and there is more than one reference to the value, a new root node is added to the heap, pointing to the same values as the original initial root node, with the exception of the one value that was to have been modified for which a new node is created.

Figure 5 shows the effects of Dennis's method on the modification of an array. The second element (b) is modified; this leads to the creation of a new value (e). Since the element belongs to a structured value (and assuming reference count > 1) a new root (B) is created that refers to the array [a, e, c]. Meanwhile, value A remains unmodified, preserving the functional semantics of the model. This method prevents copying a large amount of complex values. It also permits the sharing of identical data elements which saves memory. However this method is not the best for all situations. For example, consider a dataflow program that has two main sections. The first creates a 100-element array and populates it, one element at a time. The second takes the array and reads the elements, one at a time. In this case, the second part cannot begin to execute until the first completes its execution, even if it could read element 1 while the first part is populating element 2 and so on. A program that could potentially take as few as 101 time units to execute, takes 200 time units. These situations led to the development of *I-structures*.

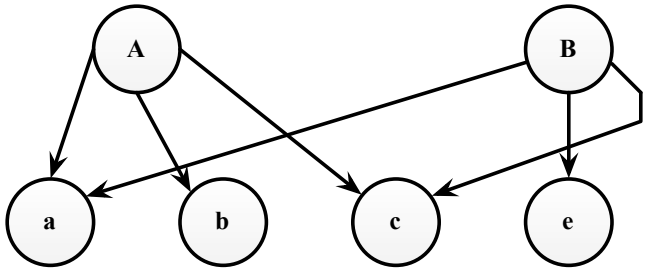


Figure 5: Dennis Method for array access

I-structures. Arvind proposed I-structures for accessing large data structures in a dataflow paradigm [5]. This method allows more flexible access to the data structures than Dennis's method. A data structure is created in memory, but its constituent fields are left blank. Each field can either store a value or be *"undefined"*. A value can only be stored in a field that is currently undefined: I-structures adhere to the single-assignment rule. Any attempt to read from an undefined field is deferred until that value becomes available (that is, until a value is assigned to the field). By following this set of rules, a data structure can be used by the rest of the program as soon as it is created while the population process continues. Meanwhile consumers can begin to read from the structure. This method has a *producer-consumer* behavior. SDF [30] uses this approach for memory management.

Hybrid scheme The idea behind the hybrid scheme [35] is to associate a template, called the *structure template*, with each array. Each array is represented by a structure template and a vector of array elements. A template is divided into the following fields:

- reference count field; used to keep track the number of hybrid structures created and destroyed during program execution;
- two pointers to two different arrays;
- location field; a string of 1's and 0's where the length of the string equals the number of elements in the array. Each location bit determines in which vector (*"left"* or *"right*") the desired element resides;
- status bit (S); it used when accessing array elements.

Whenever an array is defined, a hybrid structure is created: the elements of the array are stored sequentially in its vector, the reference count is set to one; the status bit is initialized to zero; and the left pointer is linked to the vector. Since no modifications have been made, this hybrid structure is considered to represent the original array. If we want to modify only one element, this method generates a new template structure with its right pointer pointing to a vector containing only the modified element and its left pointer pointing to the original structure template. In cases where an array is entirely updated, the sharing is no longer needed and the dependence between original and modified arrays can be removed.

3.2 Id: Irvine Dataflow Language

Id is a dataflow language developed at the University of California-Irvine [4] and was designed to provide a high-level programming language for the dynamic dataflow architecture proposed by Arvind. 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 implicitly typed by the values they carry. Composite data types include both arrays and (record) structures; and elements can be accessed using either integer indices or string values defining 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, while *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. These expressions (or sub-expressions) can be executed in any order or concurrently based on the availability of input data. Loops in Id can be easily understood from the following example, which computes:

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

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

Id also uses the concept of a *new* operation 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 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 as shown in the following:

```
y ← procedure f(n)
    (if n = 0 then 1 else n * f(n - 1))
```

The procedure y now defines factorial recursively, and the procedure is invoked as: `y(3)`.

The matrix multiply routine for Id is shown in figure 6. The inputs $\boldsymbol{a}$, $\boldsymbol{b}$ are the input matrices and $\boldsymbol{l}$, $\boldsymbol{m}$, $\boldsymbol{n}$ are inputs giving the size of the matrices. Where $\boldsymbol{a}$ is $\boldsymbol{l}$ by $\boldsymbol{m}$ and $\boldsymbol{b}$ is $\boldsymbol{m}$ by $\boldsymbol{n}$. The result matrix will be $\boldsymbol{l}$ by $\boldsymbol{n}$. A new matrix is created with the *initial* statement, setting $\boldsymbol{c}$ to the null value, Λ. The *for* statement computes the rows of $\boldsymbol{c}$. The *new c[i]* creates a new matrix $\boldsymbol{c}$ from the previous matrix $\boldsymbol{c}$ by appending row $\boldsymbol{i}$ to form the new matrix. This row is created as a vector $\boldsymbol{d}$ that is initially given a null value Λ. New elements are appended to $\boldsymbol{d}$ as they are computed in the third *for* statement. The *for* statement nesting is similar to conventional imperative languages. The difference shows in the use of the *new* nomenclature to indicate a new and separate value named $\boldsymbol{s}$, $\boldsymbol{d}$, or $\boldsymbol{c}$; rather than overwriting a value in an address $\boldsymbol{s}$, $\boldsymbol{d}$, or $\boldsymbol{c}$. The inner expression from *(initial s* to *return s)* is generating and returning a scalar value to its calling expression. The intermediate expression from *(initial d* to *return d)*

is collecting a scalar from each evaluation of the inner expression and appending it to the vector ***d***. The outer expression from *(initial c* to *return c)* is collecting row vectors from the intermediate expression and appending them to the matrix ***c***. No variable is ever overwritten, each value is created and then used in the creation of the next value.

```
procedure (a, b, l, m, n)
    (initial c ← Λ
    for i from 1 to l do
        new c[i] ← (initial d ← Λ
            for j from 1 to n do
                new d[j] ← (initial s ← 0
                    for k from 1 to m do
                        new s ← s + a[i,k] * b[k,j]
                    return s)
            return d)
    return c)
```

Figure 6: Matrix Multiply in Id

No translators for converting id programs to conventional (control-flow) architectures were developed; therefore Id was used mostly by those with access to dynamic dataflow processors and Id interpreters.

3.3 VAL

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, implementation and use of arrays present unique challenges. Array bounds are not part of the type declarations In VAL. Operations are provided to find the range of indices for the declared array. Array construction in VAL is also unusual in order to improve concurrency in handling arrays. It should be noted that maintaining the single assignment feature of functional languages, traditional language syntax for accumulating values needs some changes as shown by the *eval plus a[i]* semantics in the second example below. To fully express such concurrencies, VAL provides parallel expressions in the form of *forall*. Consider the following examples:

```
forall i in [array_liml(a), array_limh(a)]
    a[i] := f(i);

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(i)* with different values of the index ***i***. In the second example we compute a single value representing the sum of the elements of the array ***a***, representing the sequential accumulation of the result. In VAL, the construction of the array elements in the first example can proceed in parallel since all functions in VAL are side-effect free. Likewise, the accu-

```
forall i in [array_liml(A), array_limh(A)],                    % range of A rows
     j in [array_liml(B[array_liml(B)]),
           array_limh(B[array_liml(B)])]                      % range of B cols
    construct
       forall k in [array_liml(B), array_limh(B)]              % range of B rows
          eval plus A[i,k] * B[k,j]                            % accumulate products
       endall
endall
```

Figure 7: Matrix Multiply in VAL

mulation in the second example also exploits some concurrency since 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 *oneof* data type, *tagcase* provides a means of interrogating values with the discriminating unions.

VAL did not provide good support for input/output nor recursion. These language restrictions provided a straightforward translation of programs to dataflow architectures, particularly static dataflow machines. The dynamic features of VAL can be translated easily if the machine supported dynamic graphs, such as the dynamic dataflow architectures.

A matrix multiplication to compute C = A * B is shown in VAL in figure 7. The use of *forall* indicates there are no interactions among the *iteration* values for $\boldsymbol{i}$ and $\boldsymbol{j}$. This allows expansion of the *forall* to be parallelized. The *array_liml* and *array_limh* built-in functions extract the lower and higher bounds of the array, respectively. Thus, $\boldsymbol{i}$ takes on all of the values of the indices for the rows of A. VAL does not have two dimensional arrays, but uses nesting to have an array of arrays. Thus, $\boldsymbol{j}$ takes on all of the values of the indices of the array that is the first row of B; which is the indices of the columns of B. The *construct* function of VAL creates a new array with a range of $\boldsymbol{i}$ where each of those elements is an array with a range of $\boldsymbol{j}$ of the base type of the A and B arrays.

Nested within the *construct* clause is another *forall* giving $\boldsymbol{k}$ the range of the rows of B (which should match the columns of A). Where the *construct* created a matrix, the *eval* function of VAL creates a single value by combining the elements accessed by the $\boldsymbol{k}$ indices. In this instance the *plus* indicates adding all of the products across the values of $\boldsymbol{k}$. Other *eval* or reduction operators include *times*, *min*, *max*, *or*, and *and*. Notice that matrix C is not explicitly declared or named. Basically, the new matrix exists after the second *endall*. If this code fragment was in a function it would return the new matrix to the calling program. Alternately, this code could have been used in the *declaration* section of a *let* block where *C :=* would precede the code and would receive the generated matrix as a named value.

3.4 SISAL

SISAL (Streams and Iteration in a Single Assignment) is perhaps the best-known dataflow language, mostly because of the support provided by the designers [23]. SISAL was heavily influenced by the VAL language. The SISAL compiler generates optimized C code as its intermediate representation and thus could be run on any platform with a C compiler. The SISAL translator and run-time support software

are available for UNIX and other platforms from http://sisal.sourceforge.net/. SISAL programs consist of one or more separately compilable units, which include a single program, plus any number of modules, and interfaces as needed. 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. SISAL supports scalar data types and structures (records, unions, arrays and streams). A stream is a sequence of values produced in order by one expression and consumed in the same order by one or more other expressions. SISAL permits the creation of new values through the *new* notation. As can be seen in figure 8 each iteration of a loop assigns a value to a new instance of the iteration control variable ***i*** using the (*new i*) semantics to respect the single-assignment rule. This program implicitly constructs 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. The example also illustrates that both the original value of ***i*** and the value of *new i* are available at the same time. One way of viewing a SISAL *for* statement is that multiple instances of the loop body are created and run in parallel with separate values for ***i***.

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

Figure 8: SISAL stream

Another important characteristic of the SISAL language is the optimization in the reduction operation using binary tree evaluations. SISAL has predefined reduction operations to evaluate sum, product, min and max of a set of values. SISAL's popularity is also due to the concept of modules and interfaces. The interface shows the function templates that are visible publicly and the module defines the implementation of the functions, providing the language with a data abstraction capability.

The ubiquitous matrix multiplication function is easily implemented in SISAL [12]. The interface to the matrix multiplication function is described as:

```
interface MatrixRoutines;
   type TwoDim = array [..,..]  of type;
   function MatMul(A, B: TwoDim returns TwoDim);
end interface;
```

The first line within the interface specification defines a new type named *TwoDim* which is an array of two dimensions where the ..,.. indicates there are two unspecified dimensions within the array and each element is of *type*, meaning any primitive SISAL type can form an array. This *parameterized* function declaration is valid for integers of any size, reals, and double precision numbers. The specification requires that the elements of A, the elements of B, and the elements of the returned array are all of the same type. The second line defines the calling sequence for the matrix multiply function. It accepts values for A and B, both of which are two dimensional arrays and returns a two dimensional array as a result. The implementation of the matrix multiply is given as:

```
function MatMul(A, B: TwoDim returns TwoDim);
   if size(A, 2) ≅ size(B, 1) then error[TwoDim]
   else array[i in 1..size(A,1), j in 1..size(B, 2):
      sum(A[i,..]  * B[..,j])]
   endif
end function
```

The first line of the function body determines if the matrices are compatible for multiplication by requiring the size of the second dimension of A to be equal to the first dimension of B. One of the features of SISAL is that every type has an error representation. New values can be immediately tested for an error, or the value can continue to be processed, while the error status is propagated through subsequent steps, allowing the locations of error processing to be controlled by the programmer. The second line creates a new two dimensional array whose first dimension is the size of the first dimension of A and whose second dimension is the size of the second dimension of B. The third line of the function body uses the SISAL sum operator to accumulate all of the *A[i,k] * B[k,j]* products as the ".." nomenclature means all elements within that dimension. Notice there are no *for* statements in the code; all *distributions* are implied and no ordering is specified. SISAL uses the concept of distribution rather than iteration. This allows the multiplication of the elements to be distributed to as many multipliers as the hardware can provide and to accumulate the products to form each new array element through a tree reduction process.

Most operations on arrays, vectors, and streams in SISAL can be performed using just the dimension extents. However, some operations, such as gaussian elimination, operate on varying subsets of one of the dimensions based on another dimension. A *for* statement is used in these instances, but is implemented as parallel, distributed loop bodies each with its local value for the *iteration variable*.

4 Historical Dataflow Architectures

In this section we will explore how a processor architecture can be designed to implement the dataflow paradigm of execution. This section describes some early attempts at dataflow architectures. In section 4.5 we will explain why these early attempts were not commercially successful. More recent designs and implementations of dataflow architectures are discussed in section 5.

4.1 Dataflow Instructions

The dataflow graph shown in figure 9 computes two simple arithmetic expressions (X+Y)*(A+B) and (X-Y)/(A+B). We can write an assembly language program for this dataflow graph using a control flow architecture as shown in figure 10 using MIPS like instructions. Now consider how the same graph can be represented using a hypothetical dataflow instruction set as shown in figure 11.

We used the same order for both versions of the code in order to make the dataflow version correlate to the MIPS version. In the dataflow version, the order of execution will depend on the arrival times of the data and is not constrained to the list order of the instructions. In the control flow instructions, each instruction is provided with operand locations for its inputs and outputs. For example the instruction *ADD R11, R2, R3* uses values currently stored in registers R2 and R3 as inputs and stores the result of the operation in register R11. In contrast, the dataflow instruction *ADD 8R, 9R* is not provided with input operands, but only the identification of the destination instruction receiving its results, namely

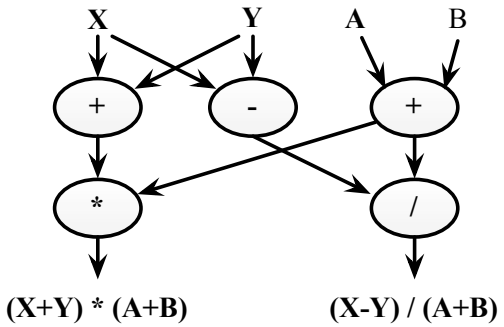


Figure 9: A simple dataflow program

```
 1.  LOAD    R2,A          ; load A into R2
 2.  LOAD    R3,B          ; load B into R3
 3.  ADD     R11,R2,R3     ; R11 = A + B
 4.  LOAD    R4,X          ; load X into R4
 5.  LOAD    R5,Y          ; load Y into R5
 6.  ADD     R10,R4,R5     ; R10 = X + Y
 7.  SUB     R12,R4,R5     ; R12 = X - Y
 8.  MULT    R14,R10,R11   ; R14 = (X+Y)*(A+B)
 9.  DIV     R15,R12,R11   ; R15 = (X-Y)/(A+B)
10.  STORE   VAL1,R14      ; store first result to VAL1
11.  STORE   VAL2,R15      ; store second result to VAL2
```

Figure 10: MIPS Like Instructions

```
 1.  INPUT   3L       ; get A, send to instr 3, left input
 2.  INPUT   3R       ; get B, send to instr 3, right input
 3.  ADD     8R,9R    ; A + B, send to instrs 8, right and 9, right
 4.  INPUT   6L,7L    ; get X, send to instrs 6, left and 7, left
 5.  INPUT   6R,7R    ; get Y, send to instrs 6, right and 7, right
 6.  ADD     8L       ; X+Y, send to instr 8, left
 7.  SUB     9L       ; X - Y, send to instr 9, left
 8.  MULT    10L      ; (X+Y)*(A+B), send to instr 10, left
 9.  DIV     11L      ; (X-Y)/(A+B), send to instr 11, left
10.  OUTPUT  VAL1     ; output first result to destination
11.  OUTPUT  VAL2     ; output second result to destination
```

Figure 11: Pure Dataflow Instructions

instructions 8 and 9 (the designation L and R with a destination are used to distinguish between left and right inputs at the destination). Thus the ADD instruction waits for its input data to arrive (in this example from instructions 1 and 2). The arrival of these inputs enables the instruction and when completed, the instruction sends the result to instructions 8 and 9.

A second difference to notice between the control flow and dataflow instructions is that in the traditional control flow architectures, instructions are executed in the order they appear. For example, instruction 7 is executed only after instruction 6 completes: a program counter keeps track of the next instruction that should be enabled for execution. In the dataflow version, no such sequence is implied. Since both instructions 6 and 7 receive their inputs at the same time (from instructions 4 and 5), they can be executed concurrently.

The goal of this example is to illustrate how we can think of designing dataflow instructions: instructions facilitate the flow of data (or results) from one instruction to another. Since in the purest form of dataflow, there are no variables or storage used, we introduced instructions such as INPUT and OUTPUT for receiving inputs or sending outputs to other modules which are outside of the graph shown. In practical implementations we will use storage for data but make sure that the data is defined only once (or apply the single assignment principle).

The various dataflow architectures differ in how to pass data among instructions, where to save inputs to an instruction while waiting for other inputs, if an instruction can be activated multiple times when multiple versions of the inputs are received (i.e., from different loop iterations).

Classical Architectures. The classical architectures that implement the dataflow model are divided into two categories: *static* and *dynamic*. The static approach allows that at most one instance of a node is executable. A node becomes executable when all the input data are available and there are no tokens on the output arc. The dynamic approach allows the simultaneous activation of multiple instances of a node: this is possible because an arc is considered as a buffer that can contain multiple data items. To differentiate the different instances of the node, a *tag* is associated with each token. The tag identifies the context in which that particular token was generated. A node is considered executable when all the inputs with the same tag are available. The static model has a simple mechanism to determine which nodes are executable, but limits the performance in case of loops, since the iterations are performed one for each time unit. The dynamic model allows a larger exploitation of parallelism: this advantage is possible thanks to the tag model but also has a significant cost in creating and managing tags (matching mechanism). The following subsections describe static and dynamic architectures.

4.2 Static Dataflow Architectures

The firing rule for a node is that a token has to be present on each input arc, and that there are no tokens present on any of the output arcs. In order to implement this, acknowledge arcs are implicitly added to the dataflow graph which go in the opposite direction to each existing arc and carry an acknowledgment token. The major advantage of the static model is the great simplicity and speed in the detection of which nodes are executable. Moreover, the memory can be allocated for each arc at compile time since each arc will contain one or zero tokens. This implies that there is no need to use complex hardware: each arc can be assigned to a particular piece of memory. Figure 12 shows the block diagram of the static architecture.

The Memory Section is a collection of memory cells. Each memory cell contains a data structure that represents an instruction. An instruction template consists of the following fields (figure 12(b)):

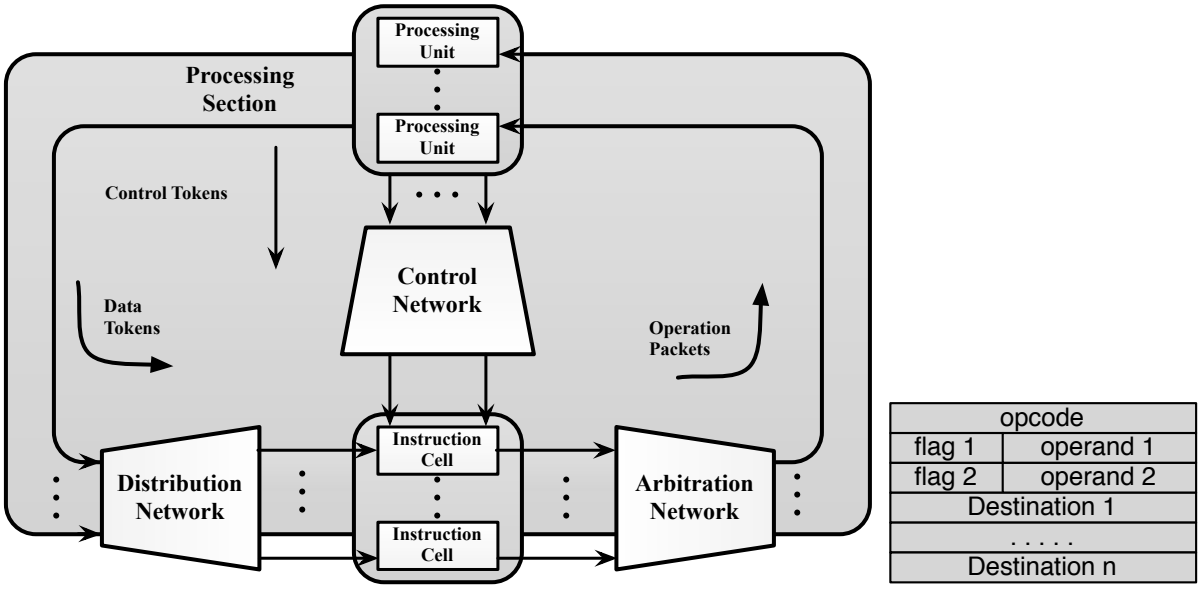


Figure 12: Static architecture(a)[23] and instruction template(b)

- opcode field;
- two fields contain the values inside tokens in the input arcs, each of which have an associated flag that indicates whether the data is available. An instruction is executed when all the flags related to inputs are set to *one* and all the flags on outputs are set to *zero*;
- a list of destination addresses for the output tokens.

The *Processing Section* is composed of 5 pipelined functional units, each of which performs operations, forms output packets, and sends the resulting token to the memory section. The *Arbitration Network* sends the data structures representing enabled instructions from the memory section to the processing section. The *Distribution Network* transfers the output packets from the processing section to the memory section. Finally, the *Control Network* reduces the load on the distribution network by transferring boolean values in *control tokens* and *acknowledge signals* from the processing section to the memory section. Control tokens convey values that are either *true* or *false*. A control token is generated by a special node, a *decider*, that applies its associated predicate to values in its input arcs. Control tokens direct the flow of data tokens by means of gates and merge actors as described in section 2.1. In spite of its simplicity, this model has many serious problems. The addition of the acknowledge arcs considerably increases the data traffic in the system (approximately 2 times). Since a node must wait for the arrival of the acknowledgment signals to be executable, the time between two successive executions increases. This may affect the performance, particularly in situations where there is not a high degree of parallelism. This model only allows an arc to contain a single token which is its most significant limitation. A subsequent iteration of a loop cannot start until the previous iteration has completed its execution even if there are no dependencies between iterations. This constraint limits the amount of concurrency that is possible in the execution within loops.

4.3 Dynamic Dataflow Architectures

The dynamic approach exposes a higher degree of parallelism by allowing multiple invocations of a subgraph as is the case with loops. Only one copy of the graph is kept in memory, and the tags are used to discern the token among different invocations. A tag is composed of the following fields:

- a field that uniquely identifies the context (instance) of a token(*invocation ID*);
- a number that identifies the iteration of the loop associated with a token (*iteration ID*);
- instruction address;
- an identifier that indicates whether the value contained in the token will be the left or right operand of the instruction (*port*).

This information is commonly known as the *color* of the token. Unlike the static model, the dynamic model represents each arc as a buffer that can contain any number of tokens, each with a different tag. In this scenario, a node is executable when the same tag is found in all its input arcs.

It is important to note that the tokens are not ordered. This may lead to an execution order that is different than the token's entrance order in the buffer. However, the tag ensures that there are no conflicts between tokens.

The tags themselves are generated by the system. The token invocation ID is unique for each subgraph. The iteration ID is initially set to zero. When a token reaches the end of the loop and it is fed back at the top of the loop, a special control operator increments the iteration ID. Whenever a token finally leaves the loop, another control operator sets its iteration ID back to zero. Figure 13 shows the block diagram of the dynamic architecture.

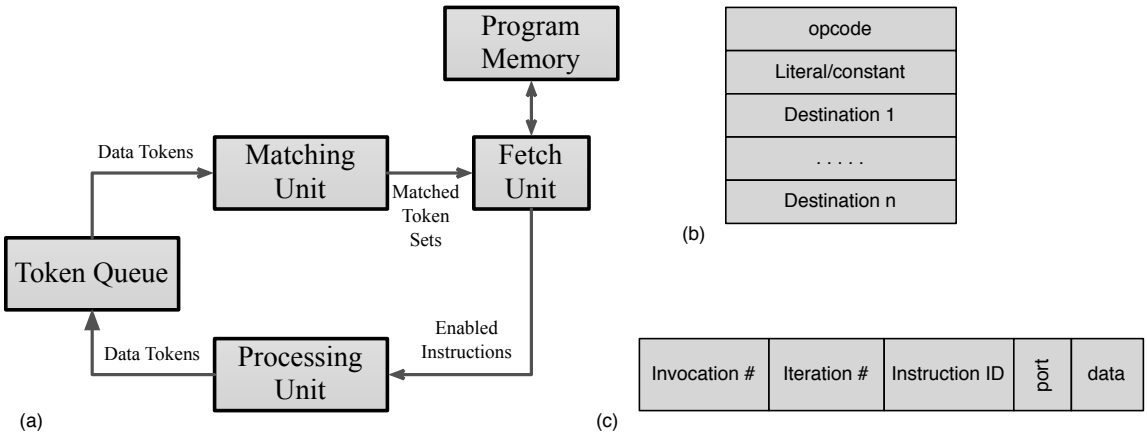


Figure 13: Dynamic architecture(a) instruction template (b) and token structure (c) [36]

The matching unit is a memory that contains the tokens waiting to be consumed. This unit groups the tokens that have the same tag. If there are tokens with the same tags, they are extracted from the matching unit and passed to the fetch unit. In the fetch unit, tags uniquely identify an instruction to be fetched from memory that contains the program. The instruction and its tokens (figure13(c)) form the

enabled instruction. The processing unit (PU) executes enabled instructions and produces the results. Then PU sends the results to the *matching unit* through the *token queue*. The main issues with the dynamic dataflow model relative to the static dataflow model are:

- overhead due to the process of comparing the tags;
- need for more memory, since each arc can contain multiple tokens.

The tag matching mechanism is the key aspect of the dataflow dynamic model. Various tests have shown that the performance directly depends on the speed of the tag matching process. It should be noted that set-associative cache memories were not available when the dynamic dataflow systems were originally designed. Existence of such technologies could have greatly simplified the matching process of dynamic dataflow architectures and increased their performance dramatically. The typical operations executed in an execution cycle are:

1. recognize the executable nodes;
2. determine the operations that must be performed;
3. compute results;
4. generate and communicate the resulting token to destination nodes.

Compared to the execution cycle of a control-flow system, these operations incur higher overheads. For example, the comparison of the tag is a more complex operation than incrementing the program counter. *Point 4* imposes an overhead which is comparable to the writing of the result in memory or in a register.

4.4 Explicit Token Store (ETS).

The problems described above have led to the introduction of the ETS model. 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 utilization of memory locations is defined at compile time while memory allocation takes place directly at runtime. A code block is represented by the pair <FP, IP>, called a *continuation*, where *FP* is the pointer to the activation frame and *IP* is set initially to the first statement in the block. A typical instruction specifies the opcode, an offset within the activation frame where it checks the availability of inputs and one or more offsets, called *displacements*, which define the instructions that will receive the result calculated by the node. Each displacement also has an associated indicator (left / right) that specifies whether the result will be the left or right operand of the destination instruction. Figure 14 shows an example of code block invocation with its instruction memory and frame memory.

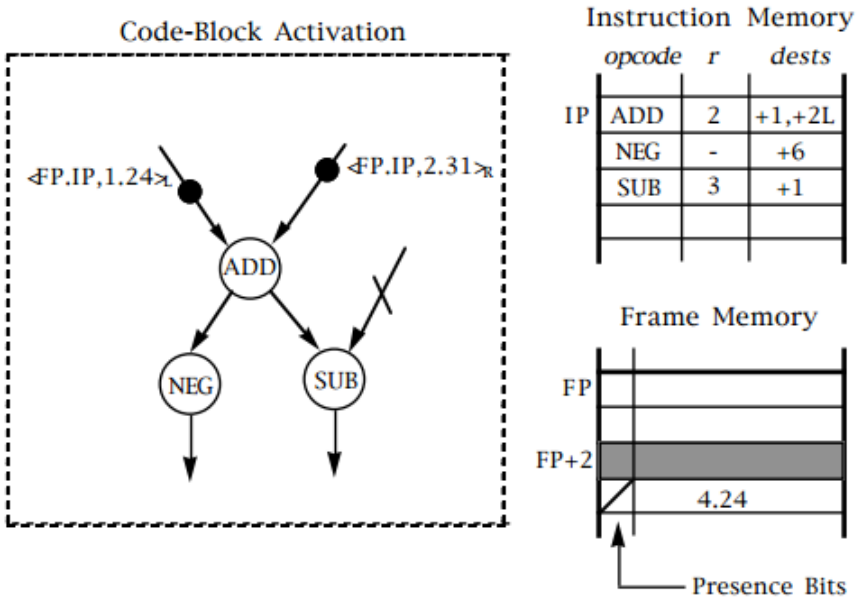


Figure 14: ETS code block[23]

When a token arrives at a node, in this case the node ADD, the IP points to the relative instruction in the instruction memory. The system executes the process of comparing the inputs in the slot that is specified by FP + r. If the slot is empty, the system writes the value in the slot and sets the presence bit to indicate that the slot is full. If the presence bit is already set, it means that one of the two operands of the instruction is already available, so the system can now execute the instruction. The steps that the system performs are:

- extract the value, leaving the slot empty and free
- perform the operation of the instruction
- communicate the result token to the destination instructions according to the displacement
- update the instruction pointer.

For example, once the ADD instruction is completed, two tokens are generated: one is directed to the instruction NEG with token <FP, IP +1, 3.55> and the other is directed to the instruction SUB with token <FP, IP+2, 3.55>.

4.5 Dataflow Limitations

After describing some of the initial implementations of dataflow systems we can now discuss some of the inherent limitations of the dataflow paradigm. While the dataflow model offers advantages in terms of

expressing concurrency and synchronization, dataflow is more difficult to implement using conventional hardware technologies. These limitations can be summarized as the following:

- Difficulty in exploiting memory hierarchies
- Too fine grained computations - each instruction is an independent activity
- Asynchronous triggering of instructions
- Implementing imperative memory system and memory ordering

Localities and Memory Hierarchy: Since pure dataflow only operates on values, the values must be communicated to functional units. Implementation of this communication can be very inefficient. Most control processor designs rely on the use of memory hierarchies including multiple levels of cache memories. Arithmetic functional units operate on registers, which are very fast devices for containing data. Control flow architectures benefit from the concept of localities in using memory hierarchies; and the localities are an artifact of control flow based sequencing (or ordering) of operations in the processor. Instructions are sequenced by using a program counter. Although branch operations may change the sequence of instructions; in most cases, it is easy to predict the location of the next instruction and data.

On the other hand, a dataflow model defines no such sequencing. Thus, it is difficult to predict which instruction will be executed next. However, reordering of instructions of such a program based on certain criteria [52] [50] can produce synthetic localities. The recurrent use of instructions (in different activation frames) also causes the existence of temporal localities. Another ordering can be created on the basis of their expected order. This can done by grouping instructions into execution levels (or E-levels [51]). Instructions that become ready (i.e., all inputs are available) at the same time unit are said to be in the same level. Instructions at level 0 for example, are ready for execution at time unit zero. Similarly, those at level 1 become ready for execution at time unit one and so on. Instruction locality can be achieved using the E-level ordering. Since the execution of an instruction may produce operands that may be destined to the instructions in the subsequent blocks, we need to prefetch more than one block of operand locations from the operand memory. We refer to these blocks as a *working set*. Block size and working set size are optimized for a given cache implementation to achieve a desired performance. While the optimum working set depends on the program, we have found that a block size of 2 instructions and working sets of 4 to 8 instructions yield significant performance improvements.

The locality for the operand cache is related to the ordering of the instructions in the instruction cache. When the first instruction in a block is referenced, the corresponding block is brought into the instruction cache. Simultaneously, operand locations for all the operands corresponding to the instructions in the *working set* of these instructions are prefetched into the operand cache. As a result of this, the operand cache will satisfy any subsequent references to the operand cache caused by the instructions within this block. Note that the operand cache block consists of a set of waiting operands or empty locations for storing the results. By prefetching, we ensure that future stores and matches caused by the execution of instructions in the block will take place in the operand cache. These principles were used to design cache memories with ETS architecture [26]

Granularity of Computations: Older dataflow architectures treated each dataflow activity as an independent thread of computation. This requires that each instruction or activity be uniquely identified causing excessive overheads for finding and associating data with instructions. In most control flow architectures; instructions are grouped into manageable entities such as functions, modules or threads. A

computation is responsible for executing such groups of instructions. In this case we can first identify the group and then use offsets to identify individual instructions within the group. Dataflow programs can also be grouped to create blocks, frames or threads, as is the case in some hybrid architectures described later in section 4.6. However, the instructions within the group still use a data driven model of execution. Other hybrid architectures, as will be described in section 4.6, that use dataflow synchronization at thread level (a thread is enabled when it receives all its inputs) and uses sequential control flow for instructions within a thread. This model is a compromise in order to fully benefit from modern technology advances, and minimize overhead in managing fine-grained threads of computations.

Synchronous Execution: In control flow, instructions are not triggered when data is available; they are executed using a program control. It is assumed that the data is already available when the instruction is scheduled. The violation of this assumption is the cause for many problems associated with parallel programs based on control flow languages. However this assumption allows for more efficient implementations. In dataflow, the data driven execution model causes asynchronous triggering of instructions. As can be seen from the previous discussions on some dataflow architectures, an instruction requiring 2 inputs will require two cycles through the processor pipeline: the first cycle simply stores the first input in some location, while awaiting the second input. The second cycle, initiated when the second input arrives, retrieves the previously saved input along with the instruction (or opcode and destinations) and executes the instruction. To be practical, dataflow architectures must move away from a data driven paradigm and use instruction driven paradigms like control flow at the execution unit level. The instruction driven model requires space for saving inputs to instructions until they are scheduled, but each instruction requires only one cycle to execute. Scheduled dataflow 5.7 relies on the instruction driven model of execution.

Memory Ordering: To be viable, dataflow processors must execute concurrent programs written using imperative languages. Assuring correct execution of concurrent programs in these languages requires adherence to the sequential consistency paradigm, which defines an order on memory accesses. Wavescalar (described in section 5.3) proposes a solution for defining memory orders within the context of tagged token dataflow architecture.

We will describe some recent architectures that have modified the pure dataflow model in order to achieve practical implementations in section.5.

4.6 Hybrid Dataflow/Controlflow Architectures

The pure dataflow model eliminates WAR and WAW dependencies and exploits the parallelism inherent in a program, but its performance is poor for applications having a low level of parallelism. Because of the problems encountered in the pure dataflow model, researchers investigated various hybrid solutions. Most of these models used an instruction clustering paradigm: various instructions are clustered together in a thread to be executed in a sequential manner through the use of a conventional program counter. The thread is enabled to run when the inputs necessary for its execution are available.

The spectrum of hybrid control-flow / dataflow systems range from simple extensions of the von Neumann paradigm to nearly pure dataflow architectures where the level of execution granularity is increased. In the pure dataflow model each machine-level instruction is seen as a thread. In a hybrid dataflow model groups of instructions form threads that are executed through control-flow. The method of grouping instructions and the average number of instructions in a group differentiates the various hybrid systems. Various projects based on this approach have shown that the dataflow model and the control-flow model are not disjoint; but are the two extremes within a spectrum of computer architectures with hybrid models

occupying the middle of the spectrum. The following paragraphs will outline some approach for hybrid architectures.

Coarse-Grain Dataflow: This approach is based on *macro instructions*, activated according to dataflow concepts. The individual instructions within the *macro instruction* are executed in a sequential manner. Systems using this approach usually separate the token control phase from the execution phase through the utilization of FIFO buffer. Dedicated microprocessors are usually used to support the execution phase.

Complex Dataflow: This technique is based on the use of machine-level complex instructions, for example vector instructions. These instructions can be implemented through pipelines as in vector systems. Structured data are accessed in block rather than by accessing the individual item. This approach differs from the I-structure where each field of a structure is accessed individually from memory. The advantage of this type of architecture is the ability to exploit the sub-instruction level parallelism, since sub-instructions can be performed in parallel (or almost) within their complex instruction. Even in this case, the process of enabling the execution and the execution itself is separated. The major difference compared to the original dataflow model is that the tokens do not carry values. The data is collected and only used inside the processing unit.

RISC Dataflow: Systems that use this approach support the execution of code written for conventional control-flow systems. The basic aspects of this type of systems are:

- conventional RISC-like instructions;
- support for multithreading with the introduction of *fork* and *join* instructions to create and synchronize threads;
- synchronization of memory accesses through the utilization of the I-structure semantic.

Threaded Dataflow: The idea behind this approach is the grouping of a certain number of instructions that are executed in a sequential manner. In particular each subgraph that exhibits a low level of parallelism is identified and converted into a sequential thread. In this way, when a thread is executed it is not necessary to perform the data availability verification process for all its instructions, but only for the first. The instructions within a thread use registers to transmit data to other instructions. These registers may be used by any instruction in the thread. The advantage of the threaded dataflow approach is that those parts of the dataflow graph that do not exhibit good potential parallelism can be executed without the associated overhead, while those that do show potential parallelism can take advantage of it. In this type of architectures, the main issue is the implementation of an efficient mechanism for synchronization between threads: ETS-like methods can be used for this.

5 Recent Dataflow Architectures

Recent improvements in computer technology (such as Network-on-a-chip and higher density integration) and limitations to instruction and thread level parallelization within conventional control flow architectures have led to a new interest in the dataflow computing model. The following sections describe some recent dataflow-based projects.

5.1 TRIPS

The TRIPS [11][18] system employs a new instruction set architecture (ISA) called *Explicit Data Graph Execution* (EDGE). Two defining features of EDGE ISA are block-atomic execution in which blocks are composed of dataflow instructions and direct instruction communication within a block. An EDGE microarchitecture maps each compiler-generated dataflow graph to a distributed execution substrate. The TRIPS ISA aggregates up to 128 instructions in a block. The block-atomic execution model logically fetches, executes, and commits each block as a single entity. Blocks communicate through registers and memory. Within a block, *direct instruction communication* delivers results from producer to consumer instructions in dataflow fashion. Figure 15 shows the TRIPS architecture.

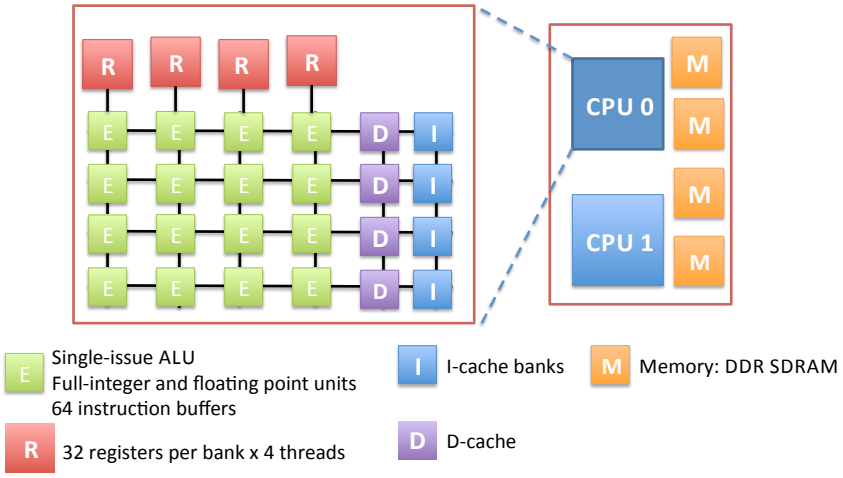


Figure 15: TRIPS microarchitecture[11]

Each TRIPS chip contains two processors and a secondary memory system, each interconnected by one or more micro-networks. Each processor contains five types of units: one global control unit (GU), 16 execution units (EU), four register units (RU), four data caches (DU) and five instruction caches (IU). The units communicate via six micro-networks that implement distributed control and data protocols. The *GU* sends a block address to the *IU* which delivers the blocks's computation instructions to the reservation stations in the 16 execution units, 4 per unit as specified by the compiler. The *IU* also delivers the register read/write instructions to reservation stations in the *RU*s. The *RU*s read values from the global register file and send them to the *EU*s, initiating the dataflow execution. The commit protocol updates the data caches and register file with the speculative state of the block. The *GU* uses its next block predictor to begin fetching and executing the next block while previous blocks are still executing. The prototype can simultaneously execute up to eight 128-instruction blocks giving it a maximum window size of 1024 instructions. On simple benchmarks TRIPS outperforms the Intel Core 2 by 10% and using hand-optimized TRIPS code, TRIPS outperforms the Core 2 by factor of 3 [18].

5.2 Data-Driven Workstation Network

D^2NOW [33] is a parallel machine based on the data-driven multithreading (DDM) model of execution. This model decouples the synchronization from the computation portions of a program allowing them to execute asynchronously; similar to the decoupling used in SDF [30, 31]. This aspect provides effective latency tolerance by allowing the computation processor to produce useful work, while a long latency event is in progress. In the DDM model, scheduling of threads is determined at runtime based on data availability: a thread is scheduled for execution only if all of its input data is available in the local memory (dataflow paradigm). Figure 16 shows a DDM processing node.

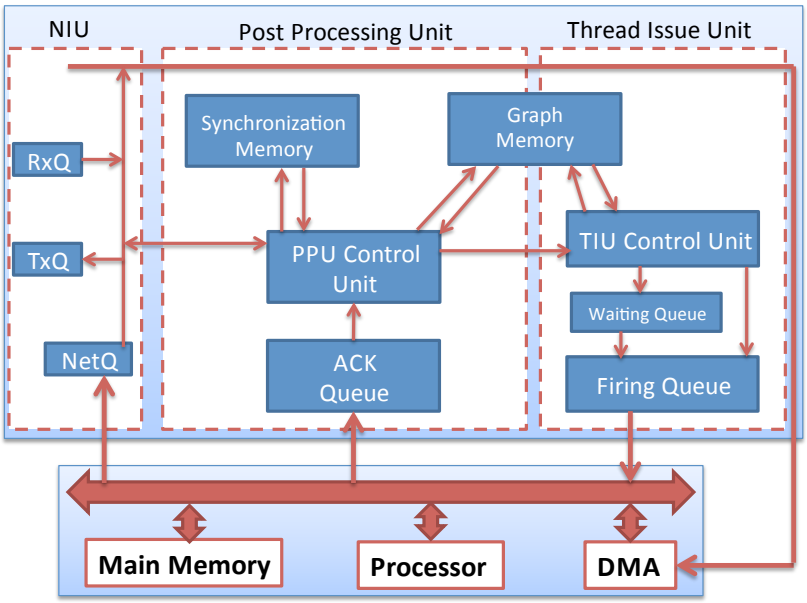


Figure 16: DDM processing node [33]

The processing node consists of a workstation augmented with the thread synchronization unit (TSU). The computation processor communicates with the TSU via two queues: the *Ready Queue* (RQ) and the *Acknowledgement Queue* (AQ). These queues are memory-mapped, there is no need for modifying the processor or adding extra instructions. The RQ contains pointers to the threads that are ready for execution. The AQ contains identification information and status of executed threads. The main storage units in the TSU are the *Graph Memory* (GM) and the *Synchronization Memory* (SM). The GM contains the informations related to the instructions, to the data memory and to the consumer threads. The SM contains the number of input data items that are necessary for a thread. The processor reads the address of the next thread to be executed from the RQ. After the processor completes the execution of a thread, it stores in the AQ the identification number and status of the completed thread and then reads the address of the next thread to be executed from the RQ. The control unit of the TSU fetches the completed threads from the AQ, reads their consumers from GM and then updates the counter of input data needed of the corresponding consumer threads in the SM. If any of these consumers are ready for execution, they are placed in the RQ and wait for their turn to be executed.

D^2NOW is built using Intel Pentium microprocessors connected through a fine-grained interconnection network. The most interesting aspect of this architecture is the communication mechanism. Like the other dataflow architectures the main problem is the overhead due to the communication between scheduling quantums (in this case, threads). In order to tolerate the communication latency, the communication is classified into three types: *fine-grain* (one data value per message); *medium-grain* (up to a few tens of values per message); *coarse-grain* (hundreds to thousands of values per message). In the fine-grain mechanism all the information is passed to the NIU (see figure 16). In the second mechanism the computation processor stores data in a buffer within the NIU, the consumer thread's identification as well as the starting address and size of the memory block. In the background, an embedded processor together with a DMA engine process the communication though the fine-grain interconnection network. The coarse-grain mechanism utilizes an Ethernet network to support large block data transfers. In this way the latency due to the large amount of data does not affect the transfers on the fine-grain network. For 16 and 32-node machines, the observed speed up, compared to the sequential single node execution, is 14.4 and 26 respectively [33].

5.3 Wavescalar

WaveScalar [49, 48] is a tagged-token, dynamic dataflow architecture. The goal of this architecture is to minimize the communication costs by ridding the processor of long wires and broadcast networks. To this end, it includes a completely decentralized implementation of the *token-store* dataflow architecture (section 4) and a distributed execution model. WaveScalar supports the execution of code written in general purpose, imperative languages (C , C++ or Java). Conceptually, a WaveScalar binary is a dataflow graph of an executable and resides in memory as a collection of *intelligent* instruction words. Each instruction is intelligent, because it has a dedicated functional unit. In practice, since placing a functional unit at each word of instruction memory is impractical, an intelligent instruction cache, the *WaveCache*, holds the current working set of instructions and executes them in place. The WaveScalar compiler breaks the control flow graph of an application into pieces called *waves*. Figure 17 shows a loop divided into waves.

The *wave-number* specifies different instances of specific instructions. Waves are very similar to *hyper-blocks*, but they are more general, since they can contain control flow joins and can have more than one entrance. Consider the waves in figure 17. Assume the code before the loop has wave number 0. When the code executes, the two CONST instructions will produce *0.0* (wave number 0, value 0). The *WAVE-ADVANCE(WA)* instructions will take these as inputs and each will output *1.0*, which will propagate through the body of the loop as before. At the end of the loop, the right-side *STEER(S)* instruction will produce 1.1 and pass it back to the *WA* at the top of its side of the loop which will then produce 2.1. A similar process takes place on the left side of the graph. The execution ends when the left-side *S* produces the value *5.10*.

One of the most interesting aspects is the method used by WaveScalar to order memory operations, to assure memory consistency. Within a basic block, memory operations receive consecutive sequence numbers. By assigning sequence numbers in this way, the compiler ensures that sequence numbers increase along any path through the wave. Next, the compiler labels each memory operation with the sequence numbers of its predecessor and of the successive memory operation, if they can be uniquely determined. The combination of instruction sequence number and predecessor and successor sequence numbers form a *link*, which is denoted: *<pred, this, succ>*. Figure 18 provides an example of annotated memory operations in a wave.

The main element in this architecture is the *WaveCache* (figure 19). The WaveCache is a grid of 2048

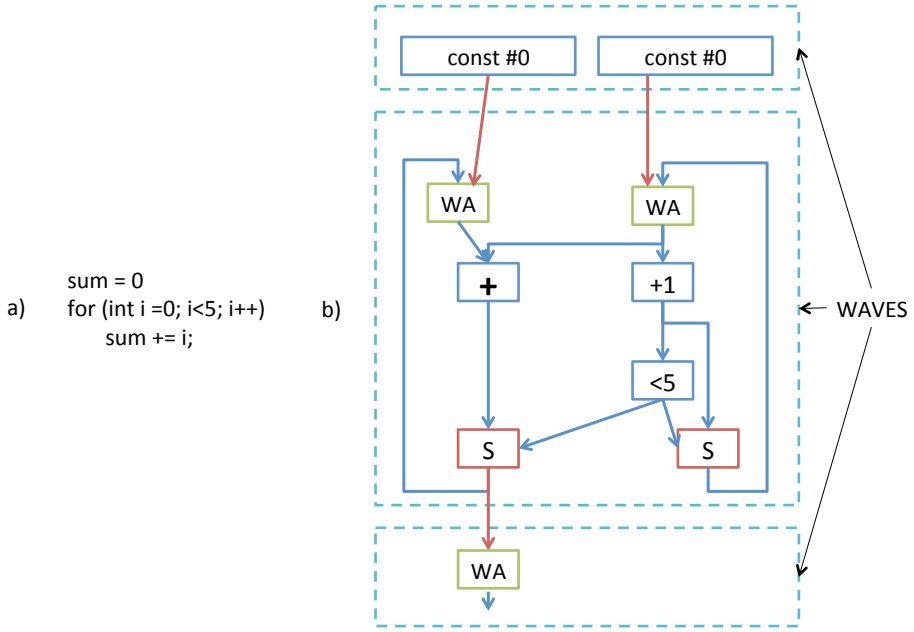


Figure 17: WaveScalar waves[49]

processing elements *(PEs)* arranged into clusters of 16. Each PE contains logic to control instruction placement and execution, input and output queues for instruction operands, communication logic, and a functional unit. Each PE contains buffering and storage for 8 different instructions. The input queues for each input require only one write and one read port and as few as 2 entries per instruction, or 16 entries total. Matching logic accesses and updates the bits as new inputs arrive, obviating the need for content addressable memories. In addition to the instruction operand queues, the WaveCache contains a store buffer and a traditional L1 data cache for each 4 clusters of PEs. The caches access DRAM through a conventional, unified, non-intelligent L2 cache.

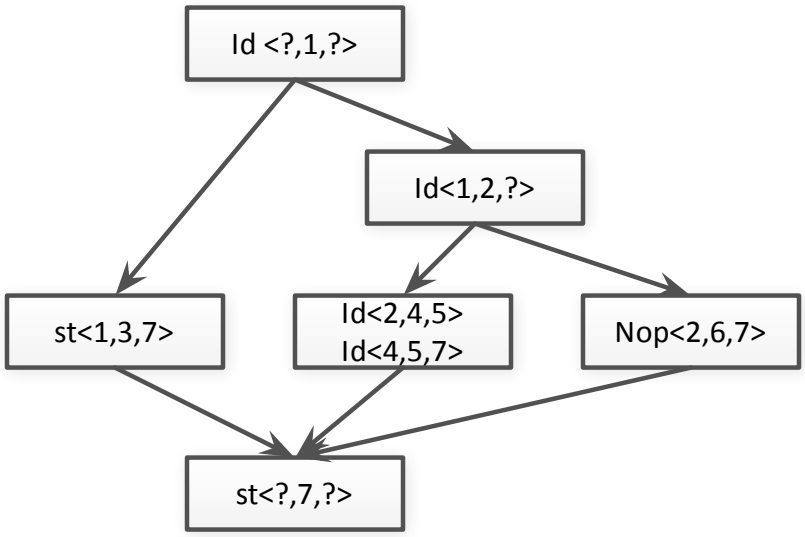


Figure 18: WaveScalar memory operations[48]

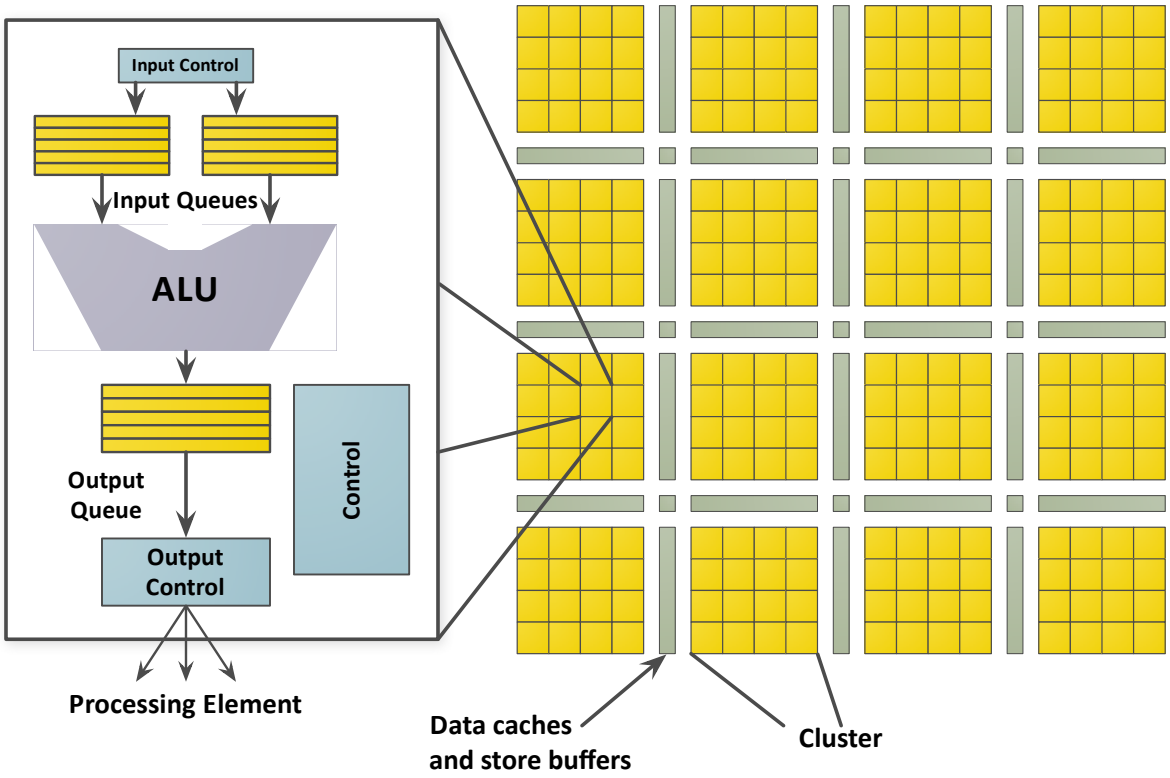


Figure 19: WaveCache[48]

5.4 TERAFLUX

Teraflux [19] is a four-year project started in 2010 within the *Future and Emerging Technologies Large-Scale Projects* funded by the European Union with the partnership of companies like HP and Microsoft, and universities in several countries. The dataflow paradigm is used at the thread level. The Teraflux system is not constrained to only follow the dataflow paradigm. The system distinguishes among legacy and system-threads (L-,S-threads) and dataflow threads (DF-threads). The execution model of Teraflux [41] relies on exchanging dataflow threads in a producer-consumer fashion. In order to support the data driven execution of threads each core includes a hardware module that handles the scheduling of threads, the *Local Thread Scheduling Unit* (L-TSU). In addition to the cores, the nodes also contain a hardware module to coordinate the scheduling of the data driven threads across nodes, the *Distributed Thread Scheduling Unit* (D-TSU). The set of all L-TSU and D-TSU composes the *Distributed Thread Scheduler* (DTS). Nodes are connected via an inter-node network, using *Network on Chip* (NoC). The following figures show the general architecture of the Teraflux system and a scheduling example.

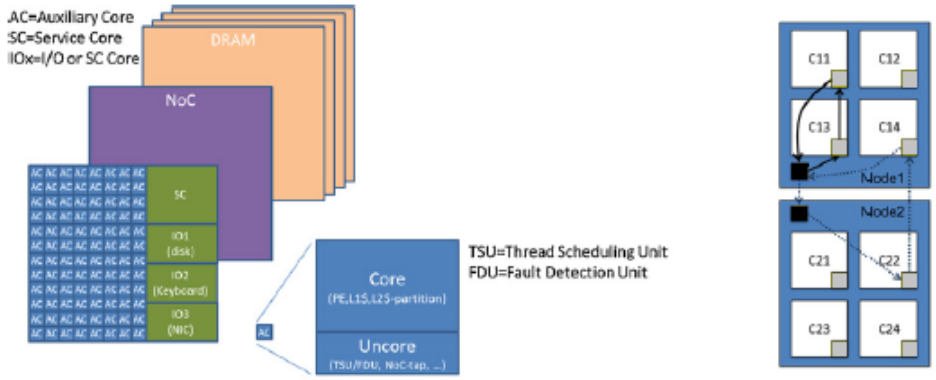


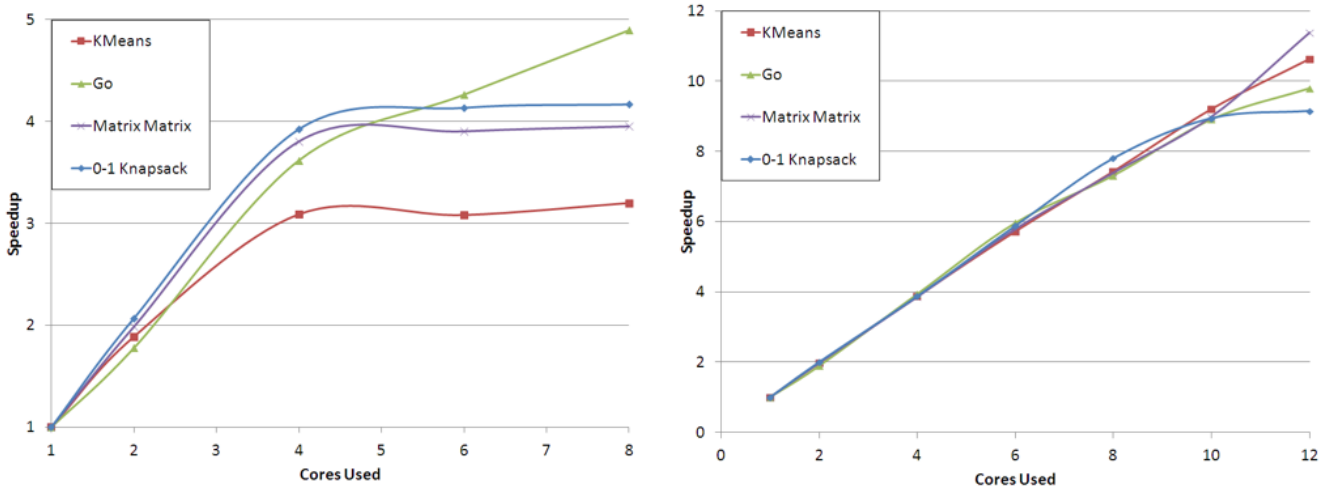
Figure 20: (a)Teraflux basic architecture and (b) Scheduling Example[41]

In figure 20(b) the L-TSU is represented in grey inside each C_{ij} core. The two D-TSUs are represented in black inside each node. Moreover the x86-x64 ISA is extended. The key points of this extension are:

- enables an asynchronous execution of threads based on dataflow rather than program control-flow;
- the execution of a DF-thread is decided by the D-TSU;
- the types of memory that are used are distinguished in 4 types (1-to-1 communication or Thread Local Storage, N-to-1 or Frame Memory, 1-to-N or Owner Writable Memory and N-to-N or Transactional Memory).

The DF-Thread scheduling policy is very similar to the one used in the Scheduled Dataflow architecture discussed in section 5.7. Since Teraflux is an ongoing project no performance data is available for the completed system. Performance data is available for DFScala [20], a library for constructing and executing dataflow graphs in the Scala [1] language. DFScala has been constructed as part of the Teraflux project

[1] http://www.scala-lang.org/



Benchmark performance on a 4 core Intel Xeon processor with hyper threading

Benchmark performance on dual 6-core AMD Opteron processors

Figure 21: DFScala performace[20]

and has been tested on Intel and AMD based systems. Figure 5.4 shows the speedup relative to a single threaded solution.

5.5 MAXELER

Maxeler Technologies is a company specializing in the development of "*dataflow computing*", a class of special-purpose computers that are programmable and can be specialized to different applications at different points in time.[40] The Maxeler system contains x86 CPUs attached to dataflow engines (DFEs) via a second-generation PCI Express bus. Data is streamed from memory onto the Maxeler DFEs, where operations are performed and data is forwarded directly from one function unit (a dataflow core) to another as the results are needed, without ever being written to the off-chip memory until the chain of processing is complete. There is no need for instruction-decode logic with this configuration as the flow of data is predetermined by the functionality programmed into the DFEs. The Maxeler dataflow machine does not require techniques such as branch prediction as these cores execute pure dataflow, or out-of-order scheduling as there are no instructions to schedule. As data is always available on chip, general-purpose caches are not needed. The minimum amount of buffering memory is utilized automatically as necessary. The Maxeler architecture and use is discussed in detail in a later section of this journal issue.

5.6 Codelet

The *Codelet* [53] project is based on the exploration of a *fine-grain, event-driven* model in support of the operations executed by high-core-count and extreme parallelism machines. This model breaks applications into *codelets* (small bits of functionality) and *dependencies* (control and data) between these objects. It then uses this decomposition to accomplish advanced scheduling, to accommodate code and data motion within the system, and to permit flexible exploitation of parallelism in support of goals for performance

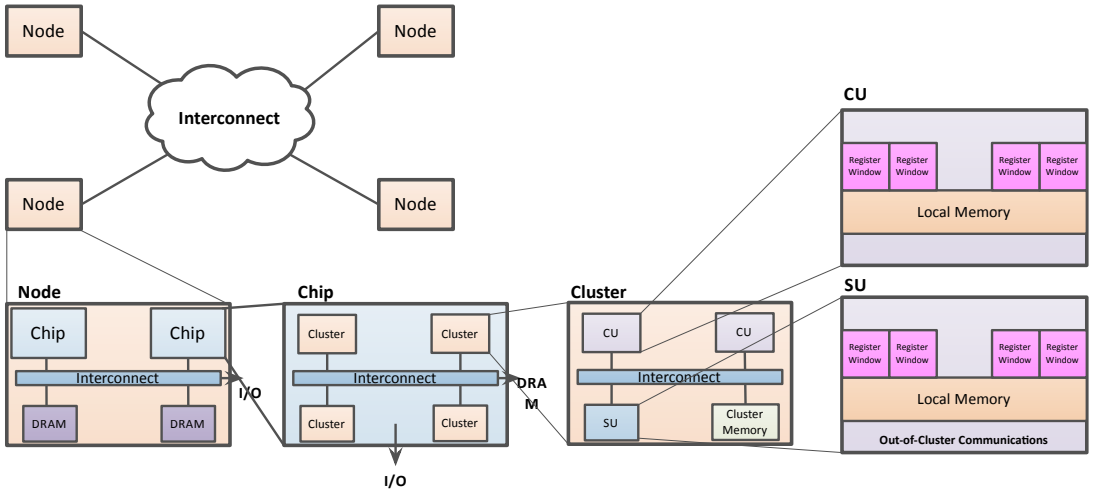


Figure 22: Codelet abstract architecture[53]

and power. Figure 22 shows the abstract machine used to explain the execution model.

The abstract machine consists of many nodes that are connected via an interconnection network. Each node contains a many-core chip. The chip may consists of up to 1-2 thousand processing cores being organized into groups (clusters) linked together via a chip interconnect. Each group contains a collection of computing units (CU) and at least one scheduling unit (SU), all brought together by their own chip interconnect.

The building blocks of an application are *Thread Procedures* and *Codelets.* A codelet is a collection of machine instructions that can be scheduled as a unit of computation. When a codelet has been allocated and scheduled to a computation unit, it will be kept usefully busy until its completion. Since codelets are more fine grained than a conventional thread, context switching is more frequent resulting in more overhead for the application.

A codelet becomes enabled once tokens are present on each of its input arcs. An enabled codelet actor can be fired if it has acquired all the required resources and is scheduled for execution. A codelet actor behaves as a dataflow node.

A threaded procedure (TP) is an asynchronous function which acts as codelet graph container. A TP serves two purposes, first it provides a naming convention to invoke codelet graphs (similar to a conventional dataflow graph), and second, it acts as a scope for codelets to efficiently operate within. Figure 23 shows three threaded procedure invocations.

5.7 Scheduled Dataflow

Scheduled Dataflow architecture [30, 31] applies the dataflow execution model at thread level rather than at instruction level, unlike conventional dataflow architectures. SDF is a multithreaded hybrid control-flow/dataflow architecture:

- the *dataflow model* is used to enable the threads for executing;
- the *control-flow* model in used to schedule instructions within a thread using a *program counter.*

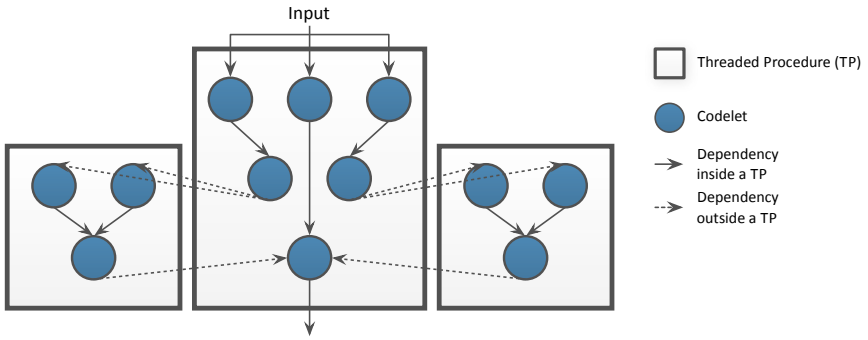


Figure 23: Codelets and Threaded Procedures[53]

SDF decouples memory accesses from execution. The first example of decoupled architecture was proposed by J. E. Smith [46]. Smith's architecture required a compiler to explicitly slice the program into memory access slices and computational slices. The two slices execute on different processing elements and communicate through buffers. A synchronization mechanism is used to guarantee the correct execution. In contrast, SDF divides a thread into three phases: the first phase pre-loads the input data from memory into registers; the second phase computes the thread results; the third phase stores the results into memory. Memory access operations (first and third phase) are executed on a different pipeline called *synchronization pipeline* (SP). The functional operations (second phase) are executed on the main pipeline which is called *execution pipeline* (EP). This allows the memory phases of a thread to be overlapped with the execution phases allowing SDF to reduce the impacts of the *memory wall*. The creation, firing, and termination of threads is performed by the *scheduling unit* (SU). There are multiple SP and EP units which allows multiple threads to be executing concurrently. The SDF architecture is shown in Figure 24. SDF threads can be executed speculatively and validated before being committed as shown in the *thread level speculation* (TLS) queue and commit blocks. This feature maps nicely to transactional memory systems.

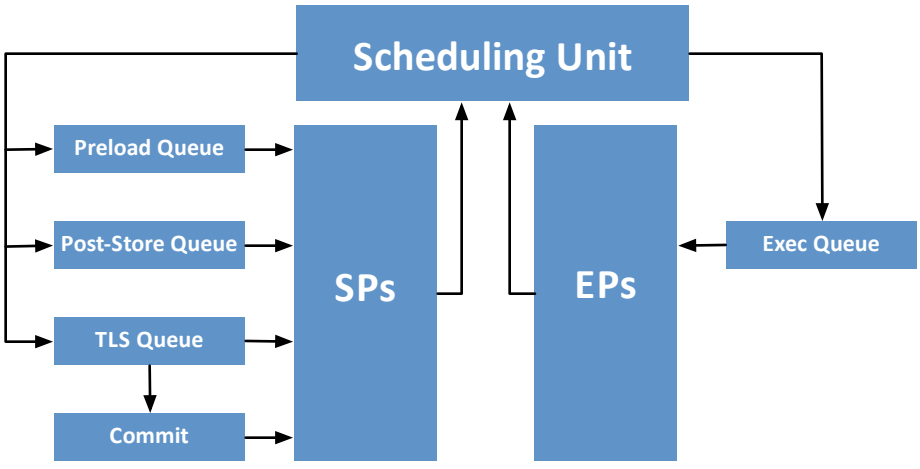


Figure 24: SDF Thread Architecture

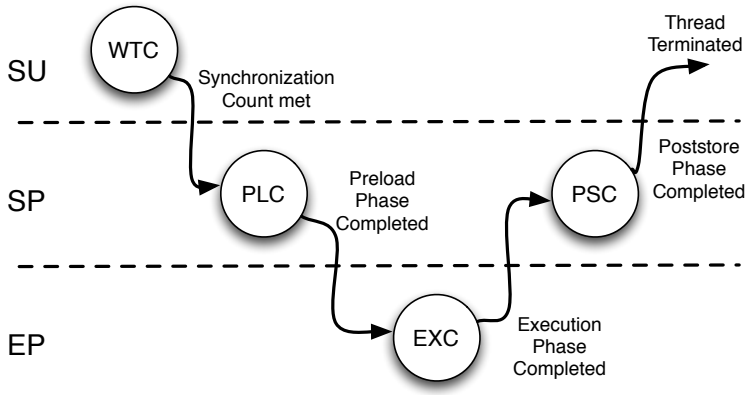


Figure 25: SDF Continuation Transitions

Threads in SDF are non-blocking. A thread completes execution without blocking the processor for synchronization with other threads. Thread context switching is controlled by the compiler, generating new threads rather than blocking a thread for synchronization. The compiler attempts to map super-blocks from the program's dataflow analysis into the SDF threads. This leads to a reduction of the synchronization overhead due to blocking threads and starvation situations are avoided. This approach generates more fine-grained threads, which may increase the overhead if not implemented in an efficient manner. SDF includes a concept of mini-threads that are useful in the execution of loops where the mini-threads execute concurrently while sharing the parent thread context, including some of the registers, for efficient loop result reduction. SDF uses ETS-like *continuation* (described in section 4.4 to maintain useful information for thread execution. Each thread has an associated continuation in one of the four following states:

1. *Waiting Continuation* (WTC): Initial state after thread creation until all inputs have been received.
2. *Pre-Load Continuation* (PLC): When a thread has received all of its inputs and is ready to be activated.
3. *EXecution Continuation* (EXC): All input data has been loaded into the registers and the thread is waiting for execution.
4. *Post-Store Continuation* (PSC): The thread has completed its computations and is ready to store the results in other threads' frames.

The movement of a thread from creation and waiting for its data to become available in the SU; to preloading its data in the SP; execution of the data operations in the EP; post-storing the results in the SP; and thread termination is illustrated in Figure 25.

5.8 Recent Architectures Summary

Table 2 provides a summary of the recent architectures discussed in the paper. The *Granularity* column indicates the general level at which dataflow techniques are applied. The *Maturity* column indicates the

maturity level of the research by listing the most complex demonstration of the architecture at the date of this paper. The *Implementation* column indicates if the research requires a full custom processor, uses a custom coprocessor, or can be implemented with existing devices. The *Programming* column indicates if special languages are needed for programming the architecture. The *Current* column indicates the currency of the research by providing the year of the most recent publication at the time this paper was written.

Architecture	**Granularity**	**Maturity**	**Implementation**	**Programming**	**Current**
TRIPS	Block	Prototype	Custom	General	2006
D^2NOW	Thread	Prototype	Existing	General	2006
Wavescalar	Superblock	Simulation	Custom	General	2010
TERAFLUX	Thread	Prototype	Existing	General	2013
MAXELER	Task	Commercial	Coprocessor	General	2013
Codelet	Thread	Abstract machine	Existing	General	2013
SDF	Superblock	Simulation	Custom	General	2013

Table 2: Summary of Recent Architectures

6 Conclusions

In this chapter we have described how dataflow properties can be used to support concurrency, synchronization, and speculation; which are essential for achieving high performance. We have provided an overview of dataflow based programming languages and historical dataflow architectural implementations. We have introduced some hybrid control flow/dataflow designs. Scheduled dataflow (SDF) architecture is used to illustrate how a dataflow architecture can be designed to exploit concurrency and speculative execution.

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. Despite the advantages of the dataflow model, there are no commercially successful implementations of the data-driven model. Critiques of dataflow argue that functional semantics and freedom from side-effects maybe nice, but well-known compiler techniques applied to imperative languages can allow equal exploitation of parallelism as dataflow. They claim that the dataflow model is weak in handling recurrences and arrays, and all proposed dataflow solutions to these problems 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.

Hybrid dataflow/control-flow architectures are proposed to address some of the these criticisms. These attempts also failed in part due to the dominance of imperative programming languages. The implementation of imperative memory systems within the context of a dataflow model is not satisfactorily addressed. 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 area.

Nevertheless, several features of the dataflow paradigm have found their place in modern processor architectures and compiler technology. Most modern processors utilize complex hardware techniques to detect data dependencies, control hazards, and dynamic parallelism to bring the execution engine closer to an idealized dataflow engine. Compilers rely on Static Single Assignment analysis to eliminate anti and output dependencies among computations. Some multithreaded systems utilize dataflow like triggering to enable threads.

There appears to be some renewed interest in hybrid systems that use dataflow like synchronization at thread level but use sequential control flow execution within a thread. Extensions to the pure dataflow model to permit "producer-consumer" and "barrier" synchronization are being proposed. Speculative execution and support for transactional memory models is yet another sign of interest in dataflow systems. But most importantly, there is a growing interest and some evidence of commercial success in using dataflow as a co-processor to a conventional control flow processing unit. The dataflow based co-processor is responsible for executing computations rich with concurrency, while the main processor will handle scheduling and memory management. A dataflow based co-processor offers the potential for a simplified alternative to conventional GPUs. We are optimistic that such systems will play an increasing role in the next generation of high performance computer systems.

References

[1] William B. Ackerman and Jack B. Dennis. Val – a value-oriented algorithmic language. Technical report, Massachusetts Institute of Technology, Feb 1979.

[2] D. A. Adams. A computation model with dataflow sequencing. Technical report, Stanford University, Dec 1968.

[3] J.M. Arul and K.M. Kavi. Scalability of scheduled data flow architecture (sdf) with register contexts. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 214–221, 2002.

[4] Arvind, Kim P. Gostelow, and Wil Plouffe. An asynchronous programming language and computing machine. Technical report, University of California, Irvine, Dec 1978.

[5] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.

[6] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990.

[7] J. Backus. Can programs be liberated from von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, pages 613–641, Aug 1978.

[8] A. Benveniste and G. Berry. The synchronous approach to reactive and realtime systems. *Proceedings of the IEEE*, 79(9):535–546, Sep 1991.

[9] Jurij Silc Borut Robic and Theo Ungerer. Beyond dataflow. *Journal of Computing and Information Technology*, 2:89–101, 2000.

[10] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Apr 1994.

[11] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, Jul 2004.

[12] D. Cann, J. Feo, W. Bohm, and R. Oldehoeft. The sisal 2.0 reference manual. Technical report, Computing Research Group, LLNL and Computer Science Department, Colorado State University, Dec 1991.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[14] J. B. Dennis. First version of dataflow procedural language. *Lecture Notes in Computer Science*, 19, 1974.

[15] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, December 1974.

[16] A. K. Deshpande and K. M. Kavi. A review of specification and verification methods for parallel programs, including the dataflow approach. *IEEE Proceedings*, 77(12):1816–1828, Dec 1989.

[17] Raphael Finkel. *Advanced Programming Languages*. Addison-Wesley Professional, 1996.

[18] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the TRIPS computer system. *SIGPLAN Not.*, 44(3):1–12, March 2009.

[19] Roberto Giorgi. Teraflux: exploiting dataflow parallelism in teradevices. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 303–304, New York, NY, USA, 2012. ACM.

[20] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján, and I. Watson. Dfscala: High level dataflow support for scala. In *Second International Workshop on Data-Flow Models For Extreme Scale Computing (DFM)*, 2012.

[21] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1319, Sep 1991.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[23] Ali R. Hurson and Krishna M. Kavi. Dataflow computers: Their history and future. In *Wiley Encyclopedia of Computer Science and Engineering*. IEEE Computer Society, 2008.

[24] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[25] G Kahn. The semantics of a simple language for parallel programming. *Proceedings of IFIP Congress Information Processing Conference*, pages 471–475, 1974.

[26] K. M. Kavi, A. R. Hurson, Patadia P., E. Abraham, and P. Shanmugam. Design of cache memories for multi-threaded dataflow architecture. In *Proceedings of the 22nd Intl. Symp. on Computer Architecture (ISCA-22)*, ISCA-22, pages 253–264, New York, NY, USA, June 1995. ACM.

[27] K.M. Kavi, B.P. Buckles, and U. N. Bhat. A formal definition of dataflow graph models. *IEEE Tr. on Comp*, pages 940–948, Nov 1986.

[28] K.M. Kavi, B.P. Buckles, and U. N. Bhat. Isomorphisms between petri nets and dataflow graphs. *IEEE Tr. on Software Engineering*, pages 1127–1134, Oct 1987.

[29] K.M. Kavi and A. K. Deshpande. Specification of concurrent processes using a dataflow model of computation and partially ordered events. *Journal of Systems and Software*, 16(2):107–120, Oct 1991.

[30] Krishna Kavi, Joseph Arul, and Roberto Giorgi. Execution and cache performance of the scheduled dataflow architecture. *Journal of Universal Computer Science, Special Issue on Multithreaded Processors and Chip Multiprocessors*, 6:948–967, 2000.

[31] Krishna M. Kavi, Roberto Giorgi, and Joseph Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Trans. Comput.*, 50(8):834–846, August 2001.

[32] Ali Hurson Krishna M. Kavi, Wentong Li. A non-blocking multithreaded architecture with support for speculative threads. *Algorithms and Architectures for Parallel Processing*, 5022:173–184, 2008.

[33] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17(10):1176–1188, October 2006.

[34] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming realtime applications with signal. *Proceeding of the IEEE*, 79(9), Sep 1991.

[35] B. Lee, A.R. Hurson, and B. Shirazi. A hybrid scheme for processing data structures in a dataflow environment. *Parallel and Distributed Systems, IEEE Transactions on*, 3(1):83–96, 1992.

[36] Ben Lee and A. R. Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, August 1994.

[37] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceeding of the IEEE*, 83(5):773–799, May 1995.

[38] Wentong Li, Krishna Kavi, Afrin Naz, and Phil Sweany. Speculative thread execution in a multithreaded dataflow architecture. In *Proceedings of the 19th ISCA Parallel and Distributed Computing Systems*, 2006.

[39] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke. Multicore compilation strategies and challenges. *Signal Processing Magazine, IEEE*, 26(6):55–63, 2009.

[40] O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, 2012.

[41] Antoni Portero, Zhibin Yu, and Roberto Giorgi. Teraflux: Exploiting tera-device computing challenges. *Procedia CS*, 7:146–147, 2011.

[42] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[43] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance scalability of decoupled software pipelining. *ACM Trans. Archit. Code Optim.*, 5(2):8:1–8:25, September 2008.

[44] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[45] Charles F. Shelor. Advances in computer architecture, principles of compiler optimization: Minithreads and dswp extensions to sdf. Technical report, University of North Texas, Computer Science and Engineering, Dec 2012.

[46] James E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, April 1982.

[47] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, May 2000.

[48] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–, Washington, DC, USA, 2003. IEEE Computer Society.

[49] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007.

[50] M. Takesue. Cache memories for data flow architectures. *IEEE Transactions on Computers*, 41:667–687, Jun 1992.

[51] S. A. Thoreson and A. N. Long. A feasibility study of a memory hierarchy in data flow environment. In *Proc. Intl. Conference on Parallel Computing*, pages 356–360, Jun 1987.

[52] M. Tokoro, J. R. Jagannathan, and H. Sunahara. On the working set concept for data-flow machines. In *Proc. 10th Annul. Intl. Symp. on Computer Architecture*, pages 90–97, Jul 1983.

[53] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.