Executing a Program on the MIT Tagged-Token Dataflow Architecture

ARVIND, SENIOR MEMBER, IEEE, AND RISHIYUR S. NIKHIL, MEMBER, IEEE

Abstract—**The MIT Tagged-Token Dataflow project has an unconventional, but integrated approach to general-purpose high-performance parallel computing. Rather than extending conventional sequential languages, we use Id, a high-level language with fine-grained parallelism and determinacy implicit in its operational semantics. Id programs are compiled to dynamic dataflow graphs, a parallel machine language. Dataflow graphs are directly executed on the MIT Tagged-Token Dataflow Architecture (TTDA), a novel multiprocessor architecture. Dataflow research has advanced significantly in the last few years; in this paper, we provide an overview of our current thinking, by describing example Id programs, their compilation to dataflow graphs, and their execution on the TTDA. Finally, we describe related work and the status of our project.**

Index Terms—**Dataflow architectures, dataflow graphs, functional languages, implicit parallelism, *I*-structures, MIMD machines.**

I. Introduction

There are several commercial and research efforts currently underway to build parallel computers with performance far beyond what is possible today. Among those approaches that can be classified as general-purpose, "multiple instruction multiple data" (MIMD) machines, most are evolutionary in nature. For architectures, they employ interconnections of conventional von Neumann machines. For programming, they rely upon conventional sequential languages (such as Fortran, C, or Lisp) extended with some parallel primitives, often implemented using operating system calls. These extensions are necessary because the automatic detection of adequate parallelism remains a difficult problem, in spite of recent advances in compiler technology [28], [2], [35].

Unfortunately, a traditional von Neumann processor has fundamental characteristics that reduce its effectiveness in a parallel machine. First, its performance suffers in the presence of long memory and communication latencies, and these are unavoidable in a parallel machine. Second, they do not provide good synchronization mechanisms for frequent task switching between parallel activities, again inevitable in a parallel machine. Our detailed technical examination of these issues may be found in [11]. In [25], Iannucci explores architectural changes to remedy these problems, inspired by dataflow architectures.

Furthermore, traditional programming languages are not easily extended to incorporate parallelism. First, loss of determinacy adds significant complexity to establishing correctness (this includes debugging). Second, it is a significant added complication for the programmer to manage parallelism explicitly—to identify and schedule parallel tasks small enough to utilize the machine effectively but large enough to keep the resource-management overheads reasonable.

In contrast, our dataflow approach is quite unconventional. We begin with *Id*, a high-level language with fine-grained parallelism *implicit* in its operational semantics. Despite this potential for enormous parallelism, the semantics are also *determinate*. Programs in Id are compiled into *dataflow graphs*, which constitute a parallel machine language. Finally, dataflow graphs are executed directly on the *Tagged-Token Dataflow Architecture* (TTDA), a machine with purely data-driven instruction scheduling, unlike the sequential program counter-based scheduling of von Neumann machines.

Dataflow research has made great strides since the seminal paper on dataflow graphs by Dennis [18]. Major milestones have been: the *U*-Interpreter for dynamic dataflow graphs [9], the first version of Id [10], the Manchester Dataflow machine [22] and, most recently, the ETL Sigma-1 in Japan [48], [23]. But much has happened since then at all levels—language, compiling, and architecture—and dataflow, not being a mainstream approach, requires some demystification. In this paper, we provide an accurate snapshot as of early 1987, by providing a fairly detailed explanation of the compilation and execution of an Id program. Because of the expanse of topics, our coverage of neither the language and compiler nor the architecture can be comprehensive; we provide pointers to relevant literature for the interested reader.

In Section II, we present example programs expressed in Id, our high-level parallel language. We take the opportunity to explain the parallelism in Id, and to state our philosophy about parallel languages in general. In Section III, we explain dataflow graphs as a parallel machine language and show how to compile the example programs. In Section IV, we describe the MIT Tagged-Token Dataflow Architecture and show how to encode and execute dataflow graphs. Finally, in Section V

Manuscript received August 5, 1987; revised February 3, 1989. This work was done at MIT Laboratory for Computer Science. This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research Contract N00014-84-K-0099. An early version of this paper appeared in the *Proceeding of the PARLE Conference*, Eindhoven, The Netherland, Springer-Verlag LNCS Vol. 259, June 1987.

The authors are with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

IEEE Log Number 8932907.

we discuss some characteristics of the machine, compare it to other approaches, and outline future research directions.

Before we plunge in, a word about our program examples. First, we are not concerned here with algorithmic cleverness. Improving an algorithm is always a possibility, but is outside the scope of this paper—we concentrate here only on efficient execution of a given algorithm. Second, even though in our research we are concerned primarily with large programs, the examples here are necessarily small because of limitations of space. However, even these small examples will reveal an abundance of issues relating to parallelism.

II. Programming in ID

We believe that it is necessary for a parallel programming language to have the following characteristics.

- It must insulate the programmer from details of the machine such as the number and speed of processors, topology and speed of the communication network, etc.
- The parallelism should be implicit in the operational semantics, thus freeing the programmer from having to identify parallelism explicitly.
- It must be *determinate*, i.e., if an algorithm, by itself, is determinate, then so should its coding in the parallel language. The programmer should not have to establish this determinacy by explicit management of scheduling and synchronization.

The last point is worth elaboration. Varying machine configurations and machine loads can cause the particular schedule for parallel activities in a program to be nondeterministic. However, the result computed should depend only on the program inputs and should not vary with the particular schedule chosen. It is a notoriously difficult task for the programmer to guarantee determinacy by explicitly inserting adequate synchronization. On the other hand, functional programming languages guarantee determinacy automatically, because of the Church–Rosser property.

Id is a high-level language—a functional programming language augmented with a determinate, parallel data-structuring mechanism called *I-structures*. *I*-structures are array-like data structures related to *terms* in logic programming languages, and were developed to overcome deficiencies in the purely functional approach (see [12] for a detailed discussion of this topic).

The exposition here relies on the intuition of the reader. The precise syntax and operational semantics of Id (expressed as rewrite rules) may be found in [34] and [13], respectively.

A. An Example Problem: Moving a Graphic Object

A graphics package requires a function to move objects around on the screen. For example, as shown in Fig. 1, we may want to "drag" a shape to a new position. A k-sided shape can be represented by a vector of k points, and a point in an n-dimensional space can itself be represented by a vector of n numbers. The distance and direction that we want the shape to move can also be represented as an n-dimensional vector. Given such a representation for a shape S and movement d, the new shape S' can be computed by simply adding vector d to each point of S. In order to explain Id, we develop the program move_shape which, given an S and a d, will produce the new shape. Along the way, we will define some functions that are useful in other contexts as well.

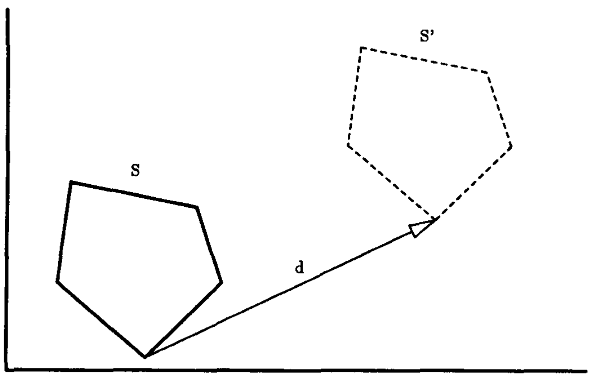

Fig. 1. Moving a shape in a two-dimensional space.

To simplify the exposition here, we assume that n is a constant, even though in Id we could discover n by querying the index bounds of, say, d. Also, we use the words "array" and "vector" synonymously.

B. Vector Sum

We begin by writing a function that moves a single point, i.e., a function that can add two vectors:

```
Def vsum A B =
    { C = array (1,n) ;
      {For j From 1 To n Do
         C[j] = A[j] + B[j]}
    In
      C } ;
```

This defines a function vsum that takes two vector arguments A and B and returns a vector result C. The body of the function is a block (the outer braces). The first statement in the block allocates the vector C with index bounds 1 to n. The second statement, the For-loop, fills it with the appropriate contents. Finally, the block's *return expression* (after the keyword In) indicates that the new vector is returned as the value of the block (which is the value of the function).

It is, of course, obvious to the reader, and perhaps can be deduced by a compiler, that the iterations of the loop are independent of each other, and hence can be done in parallel. But Id's semantics reveal much more parallelism than this. In any block, the return expression and the statements are all executed in parallel, subject only to data dependencies. Thus, the allocation of vector C can proceed in parallel with the unfolding of the loop and evaluation of all the subexpressions $A[j]+B[j]$. The array allocator returns a *descriptor* for the new vector (a pointer to memory). When C is finally available, all the pending stores $C[j] = \cdots$ can proceed.

Furthermore, the vector descriptor can be returned as the value of the function vsum even before the For-loop has terminated. This is because arrays in Id have *I*-structure semantics, eliminating read–write races. Array locations are initially empty, and they may be written at most once, at which point they become full. A reader of an array location is automatically deferred until it is full. In functional languages, a data structure whose elements can be read before all the elements of the data structure have been defined is called *nonstrict*. In this sense, all data structures in Id, including *I*-structures, are nonstrict. Generally, nonstrictness increases the opportunity for parallelism, in addition to increasing the expressive power of functional languages.

Functions can be called merely by juxtaposing them with their arguments. The expression

```
vsum e1 e2
```

represents the *application* of vsum to two arguments, the values of the expressions $e1$ and $e2$.

Functions are nonstrict in the same sense as data structures. When evaluating the function-call expression (vsum $e1$ $e2$), the output vector C can be allocated and returned, and the loop unfolded, even before $e1$ and $e2$ have produced vsum's *input* vectors A and B. The expressions $A[j]$ and $B[j]$ simply suspend until descriptors for A and B arrive. Because of this nonstrict behavior, Id can dynamically adjust to, and exploit, variations in producer-consumer (or "pipelined") parallelism, even if it depends on the inputs of the program.

We reassure the reader that the above informal explanations of the parallelism in Id will be made more precise in Sections III and IV.

C. Higher Order Functions: map_array

A very interesting and useful feature of functional languages like Id is currying, which allows us to give meaning to expressions like (vsum A). Such expressions are called *partial applications*. Suppose we write

```
move_point = vsum A ;
```

Then, the application (move_point p) is equivalent to the expression (vsum A p), and will compute a new point which is a distance A away from p. In other words, move_point is itself a legitimate unary function that adds vector A to its argument. Functions viewed in this higher order sense are said to be *curried*; they can be partially applied to their arguments, one at a time, to produce successively more specialized functions.[1]

In order to move each point of a shape, we will first write a function for the following general paradigm:

> "Do something (f) to each element of an array (X) and return an array (Y) containing the results."

This can be expressed in Id as follows:

```
Def map_array f X = { l,u = bounds X ;
                      Y = array (l,u) ;
                      {For j From l To u Do
                          Y[j] = f X[j]]
                    In
                      Y ] ;
```

Note that one of the arguments (f) is itself a function. The first statement queries the index bounds of X and binds them to the names l and u. The second statement allocates a new array Y with the same bounds. The loop fills each $Y[j]$ with the result of applying the function f to $X[j]$. The value of the block, Y, is also the value of the function.

So, to move a shape, we simply say

```
Def move_shape S d = map_array (vsum d) S ;
```

i.e., to each point in the shape S, we apply vsum d, thus computing a corresponding point displaced by d, and we collect the resulting points into an array (the result shape).

Of course, we could have written move_shape as a loop iterating over S and doing a vsum with d in each iteration. However, the recommended style for programming in Id is to use abstractions like make_array [8]. The abstractions are inexpensive—our compiler is sophisticated enough to produce code for the above program that is as efficient as one written directly using nested loops. In fact, with a handful of generally useful abstractions like map_array, one rarely needs to write loops explicitly at all. However, in this paper we use loops to minimize the gap between the source program and dataflow graphs, so that the translation is easier to understand.

D. Another Example: Inner Product

The inner product of two vectors may be written in Id as follows:

```
Def ip A B = { s = 0
             In
               {For j From 1 To n Do
                   Next s = s + A[j] * B[j]
                Finally s }} ;
```

In the first statement of the block, the value of a running sum s is bound to zero for the first iteration of the loop. During the jth iteration of the loop, the s for the next (i.e., $j+1$st) iteration is bound to the sum of s for the current iteration and the product of the jth elements of the vectors. The value of s after the nth iteration is returned as the value of the loop, block, and function.

Id loops differ radically from loops in conventional languages like Pascal. All iterations execute in parallel (after some initial unfolding), except where constrained by data dependencies. In ip, all $2n$ array selections and n multiplications may proceed in parallel, but the n additions are sequentialized.[2] The variables j and s do not refer to single *locations* which are updated on each iteration (as in Pascal); rather, every iteration has its own copy of j and s.

III. Dataflow Graphs as a Target for Compilation

In this section, we describe *dataflow graphs*, which we consider to be an excellent parallel machine language and a suitable target for programs written in high-level languages like Id. This idea was first expressed by Dennis in a seminal paper in 1974 [18]. The version we present here reflects 1) an augmentation from "static" to "dynamic" dataflow graphs that significantly increases the available parallelism [10], [9],

[1] It is to support currying notationally that parentheses are optional in function applications. For example, the curried application $f\ x\ y\ z$ would be written $(((f\ x)\ y)\ z)$ in Lisp.

[2] Of course, a different definition for ip could use a divide-and-conquer method to parallelize the additions.

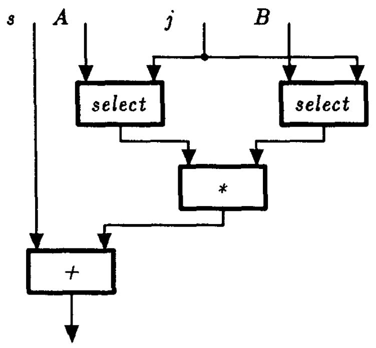

Fig. 2. Dataflow graph for $s+A[j]*B[j]$.

and 2) the introduction of numerous significant details and optimizations developed subsequently.

A. Basics

A dataflow graph consists of *operators* (or instructions) connected by directed *arcs* that represent data dependencies between the operators. Each operator may have one or more input and output arcs. Arcs may be named—the names correspond to program variables. Fig. 2 shows the graph for a simple subexpression of the inner product program *ip*.

The fork for j at the top of the figure can be regarded as a separate one-input, two-output operator, but since *any* operator can have more than one output, it would usually be incorporated as part of the preceding operator (not shown).

Data values between operators are carried on *tokens* which are said to *flow* along the arcs. In a dataflow machine, this is represented by including a *destination* in the token, that is, the address of the instruction (operator) at the end of the arc. (So, except in special signal processing architectures, one should *never* think of the dataflow graph as representing physical wiring between function modules.)

An operator is ready to *fire*, i.e., execute, when there are tokens on all its input arcs. Firing an operator involves consuming all its input tokens, performing the designated operation on the values carried on the tokens, and producing a result token on each output arc. Fig. 3 shows a possible firing sequence for our simple expression.

Tokens on the A and B arcs carry only *descriptors* (or pointers) to the I-structures themselves which reside in a memory called *I-structure storage*. (We discuss this in detail in Section III-C.) Note that the firing sequence is unspecified: operators may fire as soon as tokens arrive at their inputs; many operators may fire at the same time, and the execution times of the operators may vary.

The compilation of constants requires some care. In most cases, such as the constant 1 in the expression $j+1$, it is incorporated as an immediate operand into the + instruction itself, making it effectively a unary "+1" operator. However, if necessary, a constant can be compiled as an operator with one *trigger* input and one output (see Fig. 4). Such a situation may arise, for example, if both inputs to an instruction are constants. The data value on a trigger token is irrelevant. Whenever the trigger token arrives, the operator emits an output token carrying the constant. We discuss trigger arcs in Section III-E.

B. Functions

The body of a function definition is an expression; its dataflow graph will have

- an input arc for each formal parameter, and
- an output arc for each result.

There are two major issues to be addressed: 1) when one function *invokes* (i.e., calls) another, how should the graph of the caller be linked to the graph for the body of the callee, and 2) how to handle multiple invocations of a function that may overlap in time (due to recursion, calls from parallel loops, etc). We address the latter issue first.

1) Contexts and Firing Rules: Because of parallel invocations and recursion, a function can have many simultaneous activations. Therefore, we need a way to distinguish tokens within a function's graph that logically belong to different activations. One way to handle this would be to copy the entire graph of the function body for each activation. However, in the TTDA we avoid this overhead by keeping a single copy of the function body, and by *tagging* each token with a *context* identifier that specifies the activation to which it belongs.[3]

The reader should think of a context exactly as a "frame pointer," i.e., one should regard the set of tokens corresponding to a function activation as the contents of a frame (or "activation record") for that function. The dataflow graph for the function corresponds to its fixed code. A token carries the address of an instruction in this fixed code, and a dynamic context that specifies the frame for a particular invocation of the function. The format of a token can now be seen:

$$\langle c.s,\ v\rangle_p.$$

Here, c is the context, s is the address of the destination instruction, v is the datum, and p is the *port* identifying which input of the instruction this token is meant for. The value $c.s$ is called the *tag* of the token.[4] To simplify hardware implementation, we limit the number of inputs per instruction to two (with no loss of expressive power). Thus, p designates the "left" or "right" port. We have written p as a subscript for convenience; we will drop it whenever it is obvious from the graph.

Tokens corresponding to many activations may flow simultaneously through a graph. The normal firing rule for operators must therefore be changed so that tokens from different activations are not confused:

- An operator is ready to fire when a *matched* set of input tokens arrives, i.e., a set of tokens for all its input ports that have the same tag $c.s$.
- When the operator fires, the output value is tagged with $c.t$., i.e., the instruction in the same context that is to receive this token.

[3] Of course, this does not preclude also making copies of the function body across processors, to avoid congestion.

[4] The "tag" terminology is historical. It may be more appropriate to call it a "continuation," because it specifies what must be done subsequently with the value on the token.

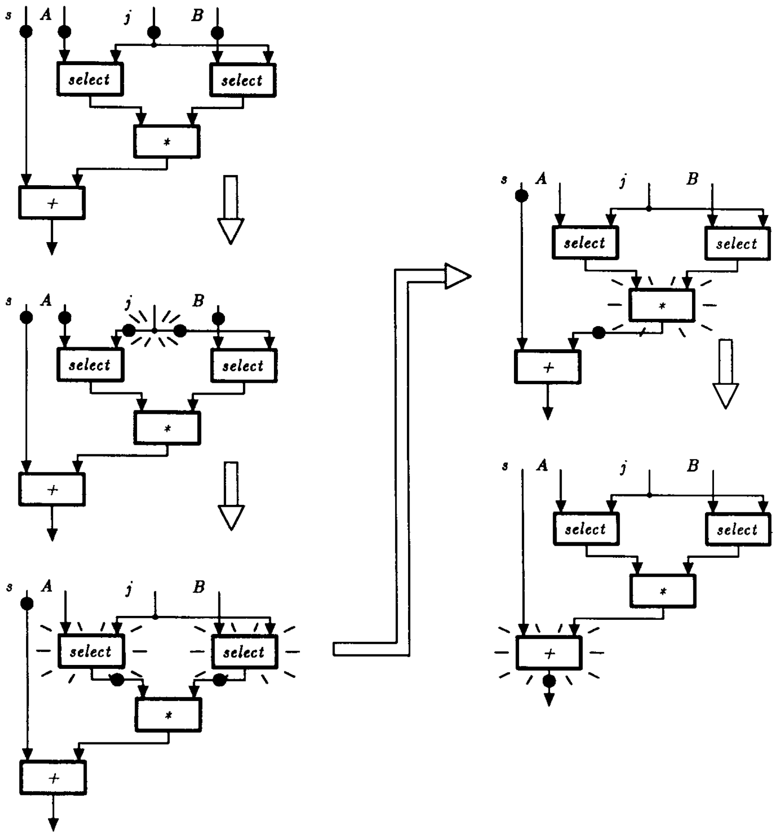

Fig. 3. A firing sequence for "$s+A[i] * B[i]$."

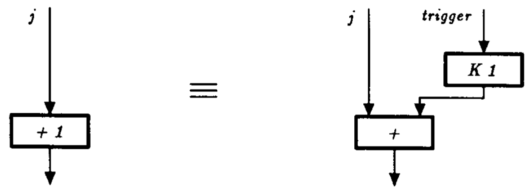

Fig. 4. Dataflow graphs for constants.

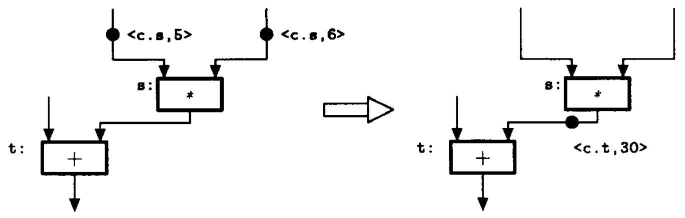

Fig. 5. Firing rule for "*" operator.

This is summarized using the following notation:

$$\text{op} : \langle c.s,\ v1\rangle_l \times \langle c.s,\ v2\rangle_r \Rightarrow \langle c.t,\ (v1 \text{ op } v2)\rangle.$$

For clarity, we will consistently follow the convention that the operator is located at address s, and its destination is located at address t. Fig. 5 shows the tag manipulation for the firing of the $*$ operator.

2) Function Linkage: In order to handle function calls, it is necessary

- to allocate a new context (i.e., a new frame) for the callee,
- for the caller to send argument tokens, including a "return continuation," to the new context, and

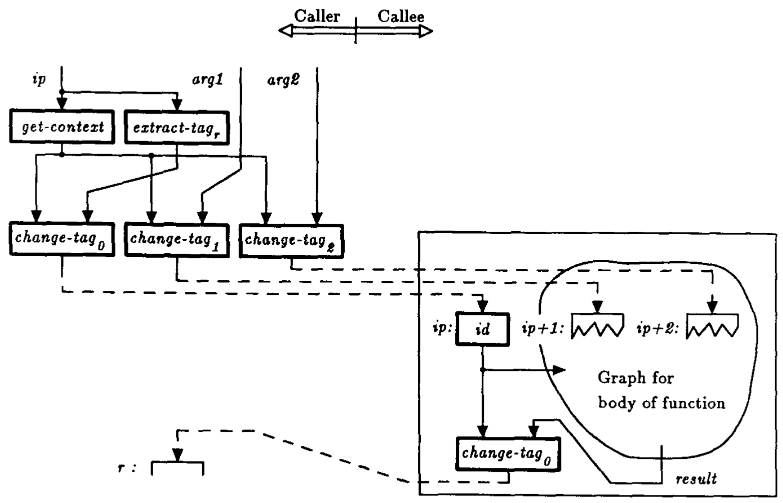

Fig. 6. Dataflow graph for function call and return linkage.

• for the callee to send result tokens back to the caller's context using the return continuation.

While reading the following description, the reader may want to refer to Fig. 6, where the graph for the function call, (ip arg1 arg2) is shown. It is assumed that r is the address of the instruction expecting the result of the function call. Thus, the return continuation is $c.r$, where c is the context of the caller. By convention, the return continuation implicitly becomes the zeroth argument. Function linkage requires instructions to manipulate contexts on tokens. The two key instructions for this purpose are extract_tag$_r$ and change_tag$_j$.

Extract_tag$_r$ is a family of monadic instructions parameterized by an address and is used by a caller to construct a return continuation for the instruction at r in the current context:

$$\text{extract\_tag}_r : \langle c.s,\ \_\rangle \Rightarrow \langle c.t,\ c.r\rangle.$$

It takes a trigger input (whose value is irrelevant) and uses the current context c to produce a tag $c.r$ as its output datum.

Change_tag$_j$ is a family of dyadic instructions parameterized by a small constant j, and is used by the caller to send arguments to the callee:

$$\text{change\_tag}_j : \langle c.s,\ c'.t'\rangle_l \times \langle c.s,\ v\rangle_r \Rightarrow \langle c'.(t'+j),\ v\rangle_l.$$

Here, v is an argument value, c' is the context of the callee, and $t'+j$ is the address of the instruction in the callee that is to receive this argument. Change_tag$_j$ is also used by the callee to send results back to the caller. In this case, v is a result value, c' is the context of the caller, and $t'+j$ is the address of the instruction in the caller that is to receive the result. Although not shown here, note that it is possible to return multiple results. By convention, the receiving instructions for multiple arguments (or results) are placed at contiguous addresses t', $t'+1$, $t'+2$, etc. Thus, for example, to send the second argument, the compiler uses a change_tag$_2$ instruction.

It is not possible to depict the output arc of change_tag graphically, because the destination of its output token is not determined statically—it depends on the left input data value. We call such arcs *dynamic arcs* and show them in figures using dashed lines.

All that remains is to allocate a new context for the callee. For this, we use the following "operator:"

$$\text{get\_context} : \langle c.s,\ f\rangle \Rightarrow \langle c.t,\ \text{new\_}c.f\rangle.$$

The input is a destination address f (the callee function's entry point), and the output is new_$c.f$, where new_c is a new, unique context identifier. The astute reader will immediately realize that there is something special about the get_context "operator." Whereas all operators described so far were purely functional (outputs depended only on the inputs), this "operator" needs some internal state so that it can produce a new unique context each time it is called. The way this is achieved is discussed in Section III-I—get_context is actually an abbreviation for a call to a special dataflow graph called a *manager*.

Now we have described the machinery used in Fig. 6 for linking function calls and returns. This linkage mechanism is only one of a number of possibilities that we have investigated.

It is important to note that the call/return scheme supports *nonstrict* functions. As suggested in Fig. 6, the zeroth argument (the return continuation) may be received by an identity instruction (*id*) that forks it and uses it as a "trigger" (to be described in Section III-E) to initiate computation in the body of the function before any of the "normal" arguments arrive. Furthermore, it is even possible for the function to return a result before the normal arguments arrive. An example of such a function is the vsum program of Section II-B, where the allocation of the result vector c does not depend on the argument vectors A and B. Thus, the part of vsum that allocates C and returns its pointer to the caller can be triggered as soon as the return continuation arrives. When the normal arguments A and B arrive, other parts of vsum will execute concurrently, filling in C's components. Our experiments show that this kind of overlap due to nonstrictness is a significant source of additional parallelism [7].

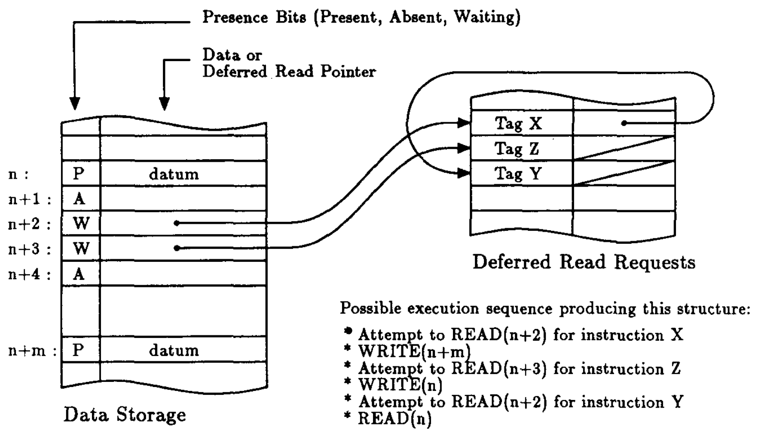
Presence Bits (Present, Absent, Waiting)
Data or Deferred Read Pointer
n : P datum
n+1 : A
n+2 : W
n+3 : W
n+4 : A
n+m : P datum
Data Storage
Tag X
Tag Z
Tag Y
Deferred Read Requests
Possible execution sequence producing this structure:
* Attempt to READ(n+2) for instruction X
* WRITE(n+m)
* Attempt to READ(n+3) for instruction Z
* WRITE(n)
* Attempt to READ(n+2) for instruction Y
* READ(n)

Fig. 7. *I*-structure memory.

C. I-structures

In the simple model of dataflow graphs, all data are carried on tokens. *I*-structures are a way of introducing a limited notion of state into dataflow graphs, without compromising parallelism or determinacy. *I*-structures reside in a global memory which has atypical read–write semantics. A token representing an *I*-structure carries only a *descriptor* of, i.e., a pointer to, an *I*-structure. When an *I*-structure token moves through a fork, only the token and not the whole *I*-structure, is duplicated, so that there can be many pointers to a structure.

A "producer" dataflow graph writes into an *I*-structure location while several other "consumer" dataflow graphs read that location. However, *I*-structure semantics require that consumers should wait until the value becomes available. Furthermore, determinacy is preserved by disallowing multiple writes or testing for the emptiness of an *I*-structure location. Even though our general discussion of TTDA architectures is in Section IV, we would like to shore up the reader's intuition about *I*-structures by presenting the *I*-structure storage model here.

1) I-Structure Storage: An *I*-structure store is a memory module with a controller that handles *I*-structure read and write requests, as well as requests to initialize the storage. The structure of the memory is shown in Fig. 7. In the data storage area, each location has some extra *presence bits* that specify its state: "present," "absent," or "waiting." When an *I*-structure is allocated in this area, all its locations are initialized to the absent state.

When a "read token" arrives, it contains the address of the location to be read and the tag for the instruction that is waiting for the value. If the designated location's state is present, the datum is read and sent to that instruction. If the state is absent or waiting, the read is *deferred*, i.e., the tag is *queued* at that location. The queue is simply a linked list of tags in the *deferred read requests* area.

When a "write token" arrives, it contains the address of the location to be written and the datum to be written there. If the location's state is absent, the value is written there and the state changed to present. If the location's state is waiting, the value is written there, its state is changed to present, and the value is also sent to all the destinations queued at the location. If the location's state is already present, it is an error.

As an aside, we would like to point out that dataflow processors with *I*-structure storage are able to tolerate high memory latencies and synchronization costs. We have given extensive reasons in [11] why it is difficult to do so in a parallel machine based on the von Neumann model.

We now return to the discussion of *I*-structures at the Id and dataflow graph level.

2) I-Structure Select Operation: The architecture takes no position on the representation of *I*-structure descriptors. One possible representation is simply a pointer to the base of the array, with its index bounds stored just below the base. In order to evaluate the expression $A[j]$, the address a to be read must be computed from the descriptor A and the index j. The address computation may also perform bounds checking (see Fig. 8). The *I-fetch*$_t$ operator then sends a "read token" to the *I*-structure storage controller with address a, along with the continuation $c.t$.

At the *I*-structure memory, if the location a has the present state, i.e., it is not empty and contains a value v, the value is sent in a token $\langle c.t, v\rangle$ to the instruction at $c.t$. If the location is in the absent state, i.e., it is empty, it is changed to the waiting state, and the continuation $c.t$ is simply *queued* at that location.

Thus, all memory reads are so-called *split-phase* reads, i.e., the request and the reply are not synchronous. The processor is free to execute any number of other enabled dataflow instructions during the memory fetch. In fact, the destination $c.t$ may be on an entirely different processor.

3) I-Structure Assignment: An *I*-structure assignment

$$A[j] = v$$

is translated into the dataflow graph shown in Fig. 9.

As in the select operation, the address a of the *I*-structure location is computed, based on the descriptor A and the index j. The *I-store* operator then sends a "write token" to the appropriate *I*-structure memory.

When the write token arrives there, the location may be in

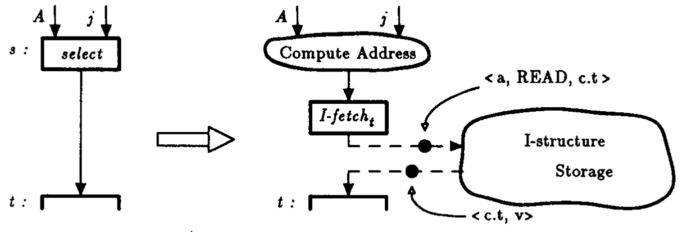

Fig. 8. Dataflow graph for *I*-structure selection.

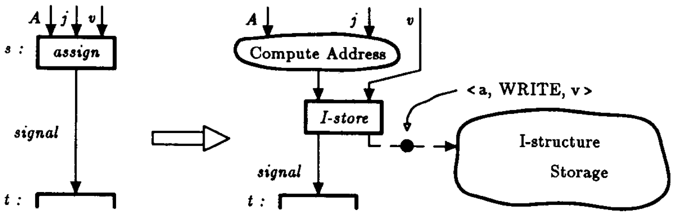

Fig. 9. Dataflow graph for *I*-structure assignment.

the absent state or in the waiting state, i.e., there are some destinations queued there from prior memory reads. The value v is written to location a, and the state is changed to present. If it was in the waiting state, a copy of v is also sent to all destinations that were queued at a.

If the location is already in the present state, i.e., it already contains a value from a previous write token, it is treated as a run-time error, since an *I*-structure location may written to at most once.

The *I-store* operator, in addition to generating the write-token $\langle a$, WRITE, $v\rangle$, also generates a *signal* token for the destination $c.t$. The signal is used, ultimately, to detect the termination of the function activation containing this *I-store* instruction. This is to ensure that the function's context is not reclaimed before all its activity has ceased. (Note that this signal does *not* imply that the actual memory write has taken place—the write token may still be on its way to *I*-structure memory.)

In some resource-management situations, it may be necessary to *know* that the write has completed at the memory unit. This can be achieved simply by doing a fetch to the same location and waiting for the response—*I*-structure semantics ensures that it cannot come back until the write has occurred.

4) I-Structure Allocation: I-structure allocation is required by the Id expression

$$\text{array } (l,u).$$

Just like the get_context "operator," we can think of a get_storage operator

$$\text{get\_storage} : \langle c.s, \text{ size}\rangle \Rightarrow \langle c.t, A\rangle$$

where size is computed from l and u, and A is the descriptor for the allocated array. Like get_context, this is also implemented by a call to a *manager* (see Section III-I). The storage allocator manager

- allocates a free area of *I*-structure memory,
- initializes all locations to the absent (i.e., empty) state, and
- sends the descriptor to the instruction at $c.t$.

Manager calls are split phase operations, like the select operation. Hence, the processor can execute other instructions while storage is being allocated.

5) Discussion: The write-once semantics that we have described supports the high-level determinacy requirements of Id. However, architecturally, and at the dataflow graph level, it is trivial to implement other memory operations as well. An "exchange" operation for managers is described in Section III-I. One could have ordinary, imperative writes as well (the storage allocator needs this). In fact, it is not difficult to include a small ALU in the *I*-structure controller to perform *fetch-and-add* style instructions [21], [41], [29].

D. Well-Behaved Graphs and Signals

When a function is invoked, some machine resources (e.g., a frame, registers) must be dynamically allocated for that invocation. We refer to these resources collectively as a *context*. Because machine resources are finite, the resources in a context must be recycled when that activation terminates. However, parallelism complicates the detection of termination. The termination of a function is no longer synonymous with the production of the result token. Because functions are nonstrict, and because there are instructions that do not return results (e.g., *I-store*), a result can be returned before all operators within the function body have executed. If resources are released before termination, there may be tokens still in transit that arrive at a nonexistent context, or worse, at a recycled context (a manifestation of the "dangling pointer" problem).

How, then, can we determine when it is safe to reclaim the resources used by a function activation? We do so by imposing an inductively-defined structure on dataflow graphs; such graphs are called *well-behaved*. We insist that all graphs have at least one input and at least one output. Then, a graph is well-behaved if

1) initially, there are no tokens in the graph;

2) given exactly one token on every input, ultimately exactly one token is produced on every output;

3) when all output tokens have been produced, there are no tokens left in the graph, i.e., the graph is *self-cleaning*.

To ensure that all our graphs are well-behaved, we construct them inductively. We start with primitive well-behaved graphs and build larger composite graphs using composition rules, or *graph schemas*, that preserve well-behavedness.

Most graph primitives are already well-behaved (+, *, $\cdots$). For some operators, such as *I-store*, it is necessary to introduce an artificial output called a *signal* to make it well-behaved. Signal tokens do not carry any meaningful values; they are used only to detect that a graph has executed.

For composite graphs, there may be many nested graphs that only produce signals. In the conditional schema (Section III-G), some data arcs may be used in one arm but not in the other. All such signals and dangling arcs are combined by feeding them into a "synchronization tree," which is a tree of dyadic synchronization operators, each of which emits a signal token on its output when it has received tokens on both its inputs. Thus, a composite graph can itself be made well-behaved by augmenting it with a suitable synchronization tree. Some examples are shown in later sections, but we gloss over many subtleties, notably signal generation for conditionals and loops; these are explained in detail in [43].

E. Code Blocks and Triggers

Apart from the common misconception that dataflow graphs represent an interconnection of hardware modules, another major misconception about dataflow is that decisions about the distribution of work on the machine are taken dynamically at the level of individual instructions. This naturally leads to fears of intolerable overheads.

The dataflow graph for a program is divided into units called *code blocks*. Each user-defined function is compiled as a separate code block. Inner loops (i.e., loops that are contained within other loops) are also compiled as separate code blocks. Of course, because of compiler transformations (such as lambda lifting [27]) and optimizations (such as in-line function expansion), there may no longer be a one-to-one correspondence between code blocks and source program functions and loops.

The "function call" mechanism described in Section III-B is, in fact, the general mechanism by which any code block invokes another. Thus, it is the code block that is the unit of dynamic distribution of work in the TTDA, at which resource allocation decisions are taken. In contrast, *within* a code block, the *work* is distributed automatically with some hardware support, as described in Section IV-C.

Every code block has one or more input arcs and one or more output arcs. One of the input arcs is designated as the trigger input and one of the output arcs the termination-signal output. When a code block B_1 invokes a code block B_2,

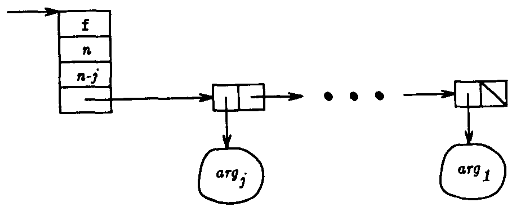

Fig. 10. Representation of a closure.

- B_1 (the caller) acquires a context for B_2 (the callee) from a manager (see Section III-I). This may involve loading code for B_2 in one or more processors.
- B_1 sends a trigger token to B_2. Usually, the return continuation token (i.e., the implicit zeroth argument) can be used as the trigger.
- B_1 sends other input tokens to B_2 (and, perhaps, continues its own execution).
- B_2 returns result(s) to B_1 (and, perhaps, continues its own execution).
- One of the "results" from B_2 is a termination signal. Often, one of the data results can be used as a termination signal.
- B_1 deallocates the context for B_2. If there is more than one output arc from B_2 back to B_1, then B_1 will need a synchronization tree to ensure that all these tokens have arrived before it deallocates B_1's context.

The top level computation of a program begins by injecting a trigger token into the outermost code block. Inner code blocks, in turn, get their triggers from their callers. The reader is referred to [43] for the details of generation and propagation of signals and triggers.

F. Higher Order Functions

Every function has a syntactically derived property called its *arity* (≥ 1) which is the number of arguments in its definition. For example, vsum has arity 2. In Section III-B, we saw how to compile expressions representing the application of a known, arity n function to n arguments [e.g., (ip arg1 arg2)]. But what about expressions where the function is applied to fewer than n arguments? (An example is the expression (vsum d) described in Section II-C.)

The "partial application" of a function of arity n to one argument produces a function that requires $n - 1$ arguments. When this, in turn, is applied to another argument, it produces a function that requires $n - 2$ arguments, and so on. Finally, when a function is applied to its last argument, the "full application," or invocation, of Section III-B can be performed.

Function values are represented by a data structure called a *closure*. Fig. 10 depicts the situation after a function f of arity n has been applied to j arguments. A closure contains:

- the entry address of the function f;
- its arity n;
- $n - j$, the number of arguments remaining;
- a list of the j argument values collected so far.

The degenerate case of a closure is the function value itself (for example, the token ip at the top left of Fig. 6); it is a

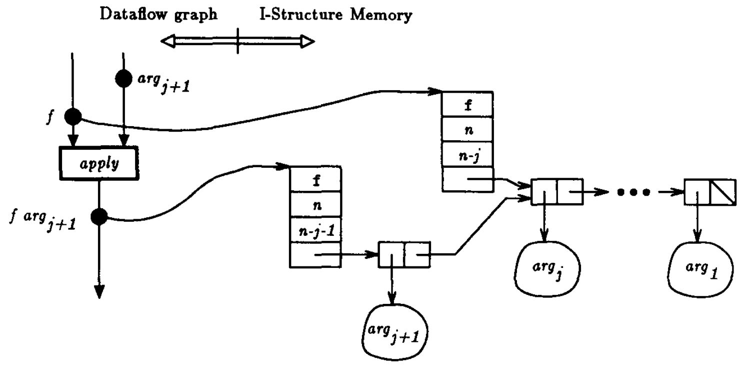

Fig. 11. Dataflow graph for partial applications.

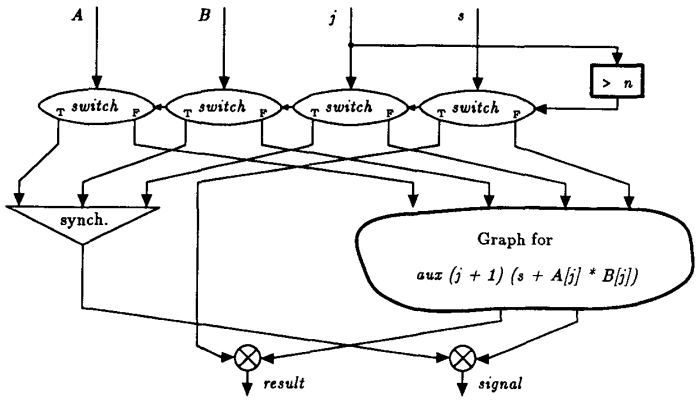

Fig. 12. Dataflow graph for a conditional.

closure with n arguments remaining and an empty list of arguments collected so far.

For general applications, we use a dyadic *apply* schema. The *apply* schema is not a primitive operator, but we will describe its behavior here without expanding it into a more detailed dataflow graph.

The left input to the *apply* schema is a closure for a function f of arity n, $n - j$ remaining arguments, and list of j collected arguments. The right input to *apply* is the next argument.

Suppose $n - j > 1$; then, this is a partial application to the $j + 1$st argument. The output of *apply* is a new closure containing the same function and arity, but with decremented arguments-remaining $(n - j - 1)$ and an augmented argument list incorporating this argument. This is depicted in Fig. 11. Note that the input and output closures *share* the first j arguments.

When $n - j = 1$, the current argument is the final argument for f, and f can now be invoked. In this case, *apply* performs a full function call, as shown in Fig. 6.

Since closures and argument-lists are implemented using *I*-structures, the *apply* schema can return the new closure even before the argument token on its right input has arrived—the allocation of the new cell in the argument list can be done immediately. When the argument finally arrives, it will be stored in the argument list. Thus, the *apply* schema is consistent with the nonstrict semantics of full function calls. The reader is referred to [43] for further details.

The general *apply* schema is of course not inexpensive. However, most applications are detectable as full-arity applications at compile time, in which case the call-return linkage is generated directly. For those familiar with the literature on compiling graph reduction, the general *apply* schema is needed only in those places where a graph *must* be constructed instead of a direct function call.

G. Conditionals

Consider the following expression (part of a tail-recursive formulation of the ip inner product function, not shown):

If $(j > n)$ Then s
Else aux $(j + 1)$ $(s + A[j] * B[j])$.

The graph for the conditional is shown in Fig. 12.

The output of the $> n$ operator is actually forked four ways to the side inputs of the four *switch* operators; the abbreviation in the picture is for clarity only.

A true token at the side input of a *switch* copies the token from the top input to the T output. A false token at the side input of a *switch* copies the token from the top input to the F output. The ⊗ node simply passes tokens from either input to its output. The ⊗ node is only a notational device and does not actually exist in the encoding of a dataflow graph—the outputs of the two arms of the conditional are sent directly to the appropriate destination. The T outputs of the A, B, and j

switches are routed to a *synchronization tree* to produce the termination signal for the true arm of the conditional.

Note that the *switch* operator is not well-behaved by itself—given a token on each of its two inputs, it produces a token on only one of its outputs. However, when used in the context of a structured conditional schema, the overall graph is well-behaved. The reader should convince himself that after a token has appeared on each of the output arcs no token could remain in the graph.

H. Loops

Loops are an efficient implementation of tail-recursive functions. In Id, the programmer may express a computation directly as a loop, or the compiler may recognize tail-recursive forms and transform them to loops.

(The impatient reader may safely skip to Section III-I, but we invite you to scan the intermediate subsection headings, hoping that you will be tempted to come back!)

We will discuss only while-loops here, using this version of the function ip which is equivalent to the for-loop version:

```
Def ip A B = { s = 0 ;
               j = 1
             In
               {While (j <= n) Do
                   Next j = j + 1 ;
                   Next s = s + A[j] * B[j]
                Finally s }} ;
```

1) Circulating Variables and Loop Constants: The body of the loop contains expressions with free variables j, s, A, and B. Two of them, j and s, are bound on each iteration using Next—we call these *circulating* variables. The dataflow graph for the loop body has an input arc and an output arc for every circulating variable. The remaining two, A and B, are invariant over all iterations of the loop, and are thus called *loop constants*. It is possible to think of loop constants as if they too, were circulating, using the trivial statements

```
next A = A ;
next B = B.
```

However, implementing them in this way would incur unnecessary overheads, and so we give them special treatment.

With every loop, we associate a region of memory in its context (frame) called its *constant area*. Before the loop body executes, there is a graph called the *loop prelude* that stores the loop constants in the constant area. Within the loop body, every reference to a loop constant is translated into a simple memory fetch from the constant area.

The dataflow graph for our loop is shown in Fig. 13. For the moment, ignore the operators labeled *D* and *D-reset*. The loop prelude stores A and B in the constant area. j and s circulate around the loop as long as the $j <= n$ output is true.

2) Loop Iteration Context: Because of the asynchronous nature of execution, it is possible that the j tokens circulate much faster than the s tokens. This means that since the loop condition depends only on j, many j tokens corresponding to different iterations may pile up on the right-hand inputs of the select operators. Thus, we need a mechanism to distinguish the

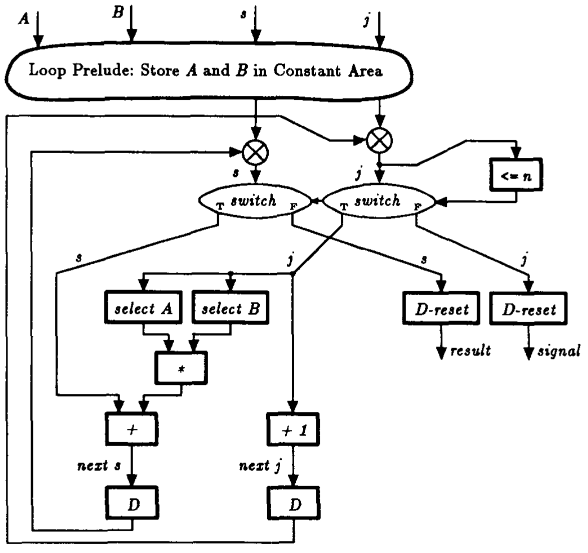

Fig. 13. Dataflow graph for a loop.

tokens corresponding to different iterations. This is performed by the *D* and *D_reset* operators. The *D* operator merely changes the context of its input token to a new context; the *D-reset* operator resets the context of its input token to the original context of the entire loop.

We could use the general get_context mechanism for this, but this would be equivalent to implementing the loop using general recursion. Instead, we assume that the get_context call that is used when invoking a code block containing a loop actually preallocates several contexts C_0, C_1, $\cdots$, for the different iterations of the loop, and returns C_0, the identifier of the first one. The structure of context identifiers is such that given the identifier C_i, we can compute the identifiers C_{i+1} and C_0. The former computation is performed by the *D* operator, which is an identity operator that simply increments the context of its input token from C_i to C_{i+1}, and the latter computation is performed by the *D-reset* operator, which is an identity operator that resets the context of its input token to C_0. The i part of the context field is called the *iteration number*. In effect, a block of resources is preallocated for the loop which is then able to perform its own resource management locally.

The astute reader will recognize that the loop-iteration-context mechanism is not sufficient to handle nested loops—tokens can still get confused. For this reason, every nested loop is packaged as a separate code block, like a procedure call, and given its own, unique context when it is invoked.

3) Loop Throttling: There remains the problem of how large should be the contiguous block of contexts for a loop. One of the problems in parallel machines is the difficulty of controlling enormous amounts of parallelism. For example, unfolding 100 000 iterations of a loop on a 256-processor machine could swamp the machine. A related point is this: in a real machine, there will be only a fixed number of bits to represent the iteration number of contexts. Hence, there is a possibility of overflow, if the loop unfolds too fast.

There is an elegant solution to these problems based on the simple observation that *all* inputs to the loop body are controlled by the bank of switch operators at the top of the loop, and these, in turn, are all controlled by a single Boolean value from the loop predicate. Any particular iteration of the loop can proceed only if the corresponding Boolean token arrives at the switches. Thus, by controlling the delivery of these Boolean tokens to the switches, we can regulate the rate at which the loop unfolds.

Suppose we wanted to limit the unfolding of a loop to some k, i.e., no more than k iterations are to exist simultaneously. The general form of a k-throttled loop is shown in Fig. 14.

The Boolean input to the switches is now *gated* using $\boxtimes$, a two-input operator that fires when both inputs arrive, copying one of them to its output. By gating this token, we can hold back an iteration of the loop. The loop prelude primes the gate with the first $k - 1$ loop iteration contexts $C_0, \cdots, C_{k-2}$, which allows the first $k - 1$ Booleans to go through, which, in turn, allows the first $k - 1$ iterations to proceed. At the bottom of the loop, each circulating variable goes through a D_k operator which increments the loop iteration context from i to $i + 1$, modulo k. Thus, the loop iteration context is the same for the ith and the $i + k$th iteration.

The reader should convince himself that tokens with contexts C_0 and C_{k-1}, inclusive, may now be sitting at the inputs to the switches. In order to prevent mismatching unrelated tokens, we must allow the C_{k-1} iteration to proceed only after the C_0 iteration is over.

The outputs of all the D_k operators are combined using a synchronization tree. When a signal token appears at the output of the tree with context C_1, we know that the C_0 iteration has terminated completely, and that there are no more C_0 tokens extant. (Recall that the loop body is itself well-behaved, by induction, so that we know that all instructions in it, including *I-stores*, have completed.) When triggered by the signal token, the D_k^{-2} operator enables the gate with a token carrying context C_{k-1} (hence the "-2" in the name, since $(k - 1) = (1 - 2) \bmod k$).

The value k may be specified as a compile-time or load-time pragma, or may be dynamically generated based on the current load on the machine. In our current graph interpreter, the user can specify it on a per loop basis at load time. There is also some code generated by the compiler, which we have glossed over, to consume the k extra tokens left at the gate when the loop terminates; for full details see [6] and [16].

Loop throttling amounts to inserting extra data dependencies in the dataflow graph. Because of this, it is possible for a throttled loop to deadlock where the unthrottled loop would not. Consider this example:

```
{ a = array (1,10) ;
  a[10] = 1 ;
  {For j From 1 To 9 Do
      a[j] = 2 * a[j+1] }}.
```

The loop unfolds forward, but the data dependencies go backward, so that $a[9]$ becomes defined first, which enables $a[8]$ to become defined, which enables $a[7]$ to become defined, and so on. If the unfolding is throttled too much (e.g., $k = 5$), the loop will deadlock, since the iteration that defines $a[9]$ cannot execute.

This example is pathological; it would have been more natural to write it with a For-downto loop instead of a For-to loop, in which case the deadlock problem does not arise. In our experience, programs rarely have dependencies that run counter to the loop direction.

To avoid deadlock, the compiler may do some analysis to choose adequately large loop bounds or to change the loop direction, but this is of course undecidable in general. The programmer also has recourse to using general recursion to avoid deadlocks due to throttled loops.

I. Managers

In any machine supporting a general-purpose programming, various resources need to be allocated and deallocated dynamically, e.g., frame and heap allocation and deallocation. We call the entities that perform these services *resource managers*. In a sequential language, such managers may not be clearly distinguishable as they are often distributed and embedded within the program itself. However, in any parallel language, these services must be shared by multiple processes (we will call them *clients*) and thus need special treatment.

Even though the bulk of a resource manager may be written as an ordinary procedure (mapping resource requests to resources), the entry and exit are handled quite differently from ordinary procedure calls. First, each resource manager must have private data structures that are shared across all calls to that manager. Second, multiple calls to a manager must be serialized so that the manipulation of these data structures is done consistently. Typically, this serialization is performed in the nondeterministic order in which requests arrive.

On a conventional machine, concurrent accesses to a manager are usually operating system calls (e.g., file allocation) and are implemented using interrupts and interrupt handlers. Serialized entry is ensured by disabling interrupts, setting semaphore locks, etc. This kind of programming is notorious for its difficulty and high probability of error. The dataflow approach offers a very clean and elegant solution to these problems, allowing significant internal parallelism within the manager itself. We present one possible implementation.

Resource managers are ordinary Id programs that run continuously and concurrently with the main application program. While no special hardware is necessary, managers do use privileged instructions that allow them to manipulate the state of the machine. For example, the I-structure memory manager uses special instructions to reset presence bits, update its memory map (such as free lists), etc.

Access to a manager is mediated by a shared *serializing queue* of requests, shown in Fig. 15. All data structures shown in the figure are located in I-structure memory. The queue is a chain of two-slot *entries*, where the first slot holds a request and the second slot points to the next entry in the queue. The second slot of the last entry (a) is empty (in the I-structure sense). The clients' interface to the manager is m, an indirection cell that always points to the last entry in the queue.

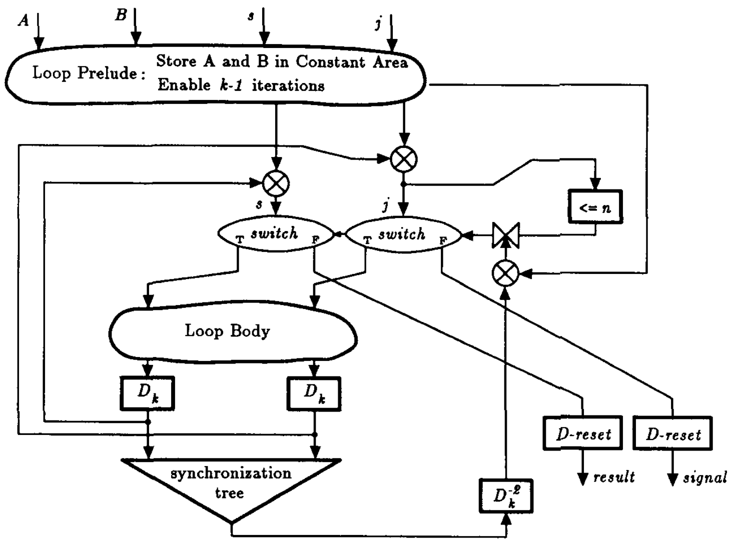

Fig. 14. Dataflow graph for throttled loops.

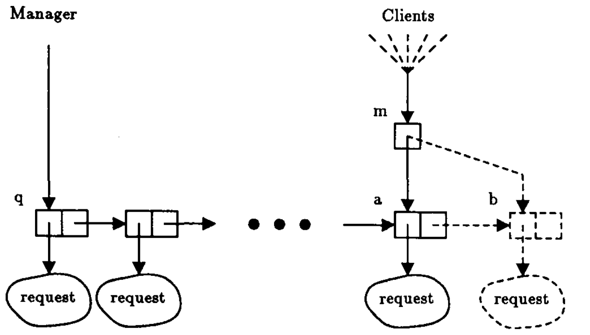

Fig. 15. Serializing input queue for a resource manager.

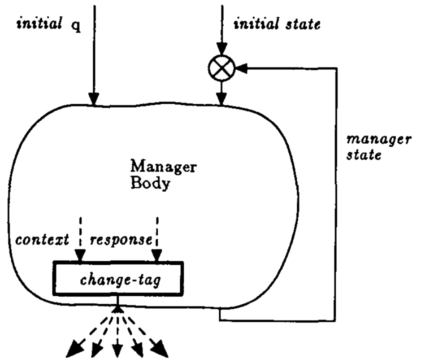

Fig. 16. Dataflow graph for a manager.

Clients use *m* to attach their requests to the end of the queue, in some nondeterministic order. The manager holds a reference to *q*, the current head of the queue. After consuming the request in the first slot of *q*, it uses the second slot to refer to the next entry in the queue, and so on. If the manager runs ahead of the requests, it suspends automatically when it tries to read the empty slot at the end of the queue.

Suppose a client wishes to send a request, such as "get a context for function ip," to the context manager. First, it *creates a new entry b*, a two-slot *I*-structure, and puts the request in *b*[1]. The client then executes (exchange *m b*), an *I*-structure operation that simultaneously fetches *a*, the current tail of the queue and stores *b*, the new tail, into *m*. The exchange is performed *atomically* at the *I*-structure memory, to ensure that two clients executing this code simultaneously do not corrupt the shared data structure. Finally, the client enqueues *b* by storing *b* in *a*[2].

The request to a manager generally contains several pieces of information, according to software convention. For example, we may have a manager exclusively for allocating and deallocating contexts (see Section III-B2). The request constructed by get_context may contain

- a request type ("allocate"/"deallocate"),
- the name of the callee function (so that the manager knows what resources to allocate), and
- a return continuation *c.t* for the managers' response (the instruction that receives the output of get_context).

Some requests, such as release_context, may not require a return continuation. The mechanism also allows other pragmatic information to be packaged with the request, such as loop bounds, priorities, etc.

Similarly, get_storage and release_storage expand into calls to a manager for heap storage, with request containing the size of the memory request.

A manager is shown in outline in Fig. 16. Each manager is initialized with an *initial state* containing data structures representing available resources and a reference to *q*, the head of its serializing queue. A request is taken off the queue, and together with the current *state*, enters the manager body (which is an ordinary dataflow graph) to actually allocate/deallocate the resource. It uses the change_tag operator and the return continuation that was packaged with the request to send the response back to the client. Finally, the manager body produces a *next state* which is fed back, ready to be combined with the next request. The behavior of a manager is abstractly modeled as follows:

```
{WHILE true DO
    request = q[1] ;
    next state = manager_body request state ;
    next q = q[2] }.
```

As pointed out earlier, the manager will automatically suspend when it tries to read the empty *I*-structure slot at the end of the queue. Using loop bounding, we can limit it to one iteration at a time. The manager body itself can have significant internal parallelism.

The manager need not respond to requests in the order in which they were received. For example, in order to favor small requests over large, a heap allocator needs the ability to defer a large request. In such cases, the manager simply stores the pending request in its state variable and examines the next request. Thus, dataflow managers permit *all* the flexibility one normally expects in resource managers, such as priority queues, preemptive resource allocation, etc.

As in any resource management system, there are some bootstrapping issues. For example, storage for the queue entry for a storage allocation request must not itself need a call to the storage manager. These issues are no different from those in conventional systems, and are handled similarly.

If all requests for a particular kind of resource (e.g., heap storage) went to a single manager, it is, of course, likely to become a bottleneck. This can be addressed in standard ways. For example, we may partition the resource into separate domains, each managed by a local manager. These managers may negotiate with each other occasionally to balance resource usage across the machine. The communication between managers is no different from the communication between parts of any other Id program.

Functional programmers will recognize that the manager queue performs a "nondeterministic" merge. However, managers are significantly easier to use than the nondeterministic merge operator, which cannot adequately cope with systems in which the users of a resource manager are dynamically determined. The reader is invited to see [4] for more details, including a programming methodology for managers.

Dataflow graphs provide all the *mechanisms* necessary to implement managers; what remains is to decide the *policies* encoded therein. This is a major area for research (see [17]), both in our project and elsewhere. Currently, a major obstacle is the general lack of experience in the research community with large, parallel applications.

IV. Dataflow Graphs as a Machine Language for the TTDA

We have seen that dataflow graphs are a good target for a compiler for a high-level programming language such as Id. Our experiments have confirmed that the tagged-token semantics for executing dataflow graphs exposes large amounts of parallelism, even in conventional algorithms [7]. In this section, we describe the MIT Tagged-Token Dataflow Architecture (TTDA), a machine architecture for directly executing dataflow graphs.

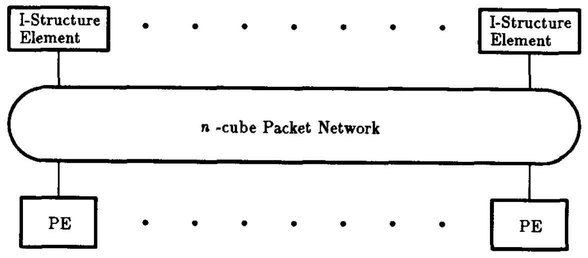

Fig. 17. Top-level view of TTDA.

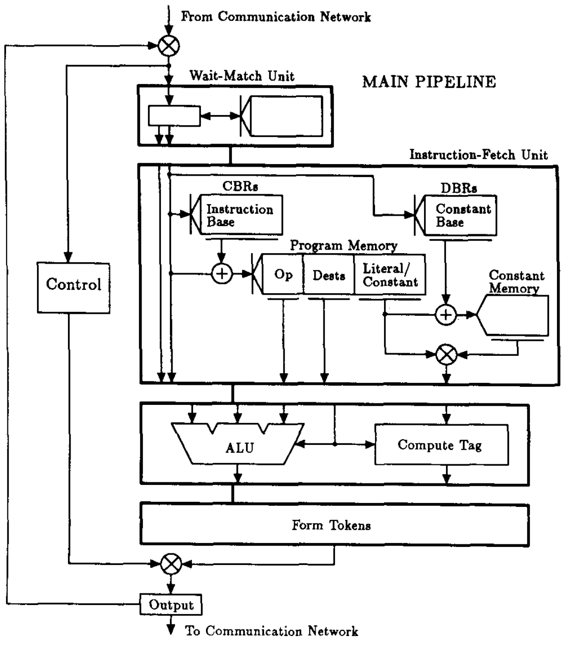

Fig. 18. A processing element.

A. Architecture

At a sufficiently abstract level, the TTDA looks no different from a number of other parallel MIMD machines (Fig. 17)—it has a number of identical processing elements (PE's) and storage units interconnected by an *n*-cube packet network. As usual, there are many packaging alternatives—for example, a PE and storage unit may be physically one unit—but we do not explore such choices here. However, it is important that the storage units are addressed uniformly in a *global address space*; thus, they can be regarded as a multiported, interleaved memory.

Each PE is a dataflow processor. Each storage unit is an *I*-structure storage unit, which was described in Section III-C. A single PE and a single *I*-structure unit constitute a complete dataflow computer. To simplify the exposition, we will first describe the operation of the machine as if it had only one PE and *I*-structure unit; in Section IV-C, we discuss multiprocessor operation.

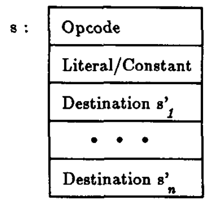

Fig. 19. Representation of an instruction.

B. Single Processor Operation

The architecture of a single processing element is shown in Fig. 18. The main pipeline is the central feature of the PE.

1) Representation of Dataflow Graphs and Contexts: Recall (from Section III-E) that a program is translated into a set of basic dataflow graphs called code blocks. The graph for a code block is encoded as a linear sequence of instructions in program memory. The assignment of instructions to addresses in the linear sequence is arbitrary, except for the call-return linkage convention mentioned in Section III-B. As an engineering decision, every instruction has no more than two inputs. Thus, every instruction in the graph is encoded as shown in Fig. 19.

The literal/constant field in an instruction may be a literal value or an offset into the constant area in constant memory. The destinations are merely the addresses of the successor instructions in the graph. To facilitate relocation, addressing within a code block is relative, i.e., destination addresses are encoded as offsets from the current instruction.

A specific invocation of a code block is determined by a *context*, which identifies two registers: a code block register (CBR) which points to the base address in program memory for the code block's instructions, and a database register (DBR) which points to the base address for the constant area in constant memory.

The get_context manager-call discussed in Sections III-B and III-I must therefore

- allocate a CBR/DBR pair and space in constant memory for the designated function,
- initialize the CBR/DBR to point to the instruction and constant base addresses, and
- return the CBR/DBR number as the context.

2) Operation of the Main Pipeline: Tokens entering the main pipeline go through the following units in sequence.

Wait-Match Unit: The *wait-match unit* (WM) is a memory containing a pool of waiting tokens. If the entering token is destined for a monadic operator, it goes straight through WM to the instruction-fetch unit. Otherwise, the token is destined for a dyadic operator and the tokens in WM are examined to see if the other operand has already arrived, i.e., if WM contains another token with the same tag. If so, a *match* occurs: that token is extracted from WM, and the two tokens are passed on to the instruction-fetch unit. If WM does not contain the partner, then this token is deposited into WM to wait for it.

The wait-match unit is thus the *rendezvous* point for pairs of arguments for dyadic operators. It has the semantics of an associative memory, but can be implemented using hashing.

Instruction-Fetch Unit: The tag on the operand tokens entering the instruction-fetch unit identifies an instruction to be fetched from program memory (via a CBR). The fetched instruction may include a literal or offset into the constant area, and a list of destination instruction offsets. The tag also specifies a DBR; using the constant base address found there and the constant offset in the instruction, any required constants from the constant area are now fetched from constant memory.

All this information—the data values from the tokens; a constant from the constant area or a literal from the instruction, the opcode from the instruction, the destination offsets from the instruction, and the context itself—is passed on to the next stage, the ALU, and compute-tag unit.

ALU and Compute-Tag Unit: The ALU and compute-tag units are actually two ALU's operating in parallel. The ALU unit is a conventional one that takes the operand/literal/constant data values and the opcode, performs the operation, and produces a result.

The compute-tag unit takes the CBR and DBR numbers from the context and the instruction offsets for the destinations, and computes the tags for the output of the operator. Recall that the tag for two instructions in the same code block invocation will differ only in the instruction offset.

The ALU result and the destination tags are passed to the form-tokens unit.

Form-Tokens Unit: The form-tokens unit takes the data value(s) from the ALU and the tags from the compute tag and combines them into result tokens.

Output tokens emerging at the bottom of the pipe are routed according to their destination addresses: back to the top of the PE, or into the network to the *I*-structure unit, or, in a multiprocessor, to other PE's and *I*-structure units. The global address space makes this routing straightforward.

The main pipeline can be viewed as two simpler pipelines—the wait-match unit and everything below it. Once tokens enter the lower pipeline, there is nothing further to block them. Pipeline stages do not share any state, so there are no complications such as reservation bits, etc.

3) The Control Unit: The control unit receives special tokens that can manipulate any part of the PE's state—for example, tokens that initialize code-block registers, store values into constant memory and program memory, etc. These tokens are typically produced by various managers such as the context manager.

The control section also contains any connection to the outside world, such as input-output channels.

C. Multiprocessor Operation

In a multiprocessor machine, all memories (program, constant, *I*-structure) are globally addressed. Thus, one can implement any desired form of interleaving. The simplest distribution of code on the TTDA is to allocate an entire code block to a PE. However, the TTDA has a sophisticated *mapping* mechanism that also allows a code block to be allocated *across* a group of PE's, thereby exploiting the internal parallelism within them. For example, it is possible to load a copy of a loop procedure on several PE's, and distribute different iterations of the loop on different PE's according to some prespecified hash function on the tags. The hash function to be used is stored in a MAP register which like the CBR and DBR is loaded at the time of procedure activation. In fact it is also possible to execute a code block on several PE's by loading only parts of it on different PE's. When a token is produced at the output of either a PE or an *I*-structure unit, its tag specifies exactly which PE or *I*-structure unit it must go to. The token is sent there directly, i.e., there is no broadcasting.

It is important to note that the program does not have to be recompiled for different mapping schemes. The machine code (dataflow graph) is independent of the mapping schemes and the machine configuration. Furthermore, the number of instructions executed does not vary with the machine configuration.

D. Discussion

It is important to realize that the PE architecture shown here is *the* hardware interpreter for dataflow graphs—it is not an abstraction to be implemented at a lower level by a conventional processor.

Any parallel machine must support mechanisms for fast synchronization and process switching. The TTDA supports these at the *lowest* level of the architecture, without any busy-waiting. Every wait–match operation and every *I*-structure read and write operation is a synchronization event directly supported in hardware. Sequential threads may be interleaved at the level of individual instructions. Unlike von Neumann machines, switching between threads does not involve any overheads of saving and restoring registers, loading and flushing caches, etc. All memory reads are *split phase*, i.e., between the request-to-read and the datum-reply there may be an arbitrary number of other instructions executed. Thus, the latency (roundtrip time) of memory reads, which is bound to be high in a parallel machine, is masked by doing useful work in the interval; the overall throughput of the interconnection network is more critical than its latency.

V. Comparison to Other Work

An invariant in our approach to the problem of high-speed, general-purpose parallel computing has been the belief that it cannot be solved at any single level. The goal will be achieved only with synergy between the language, the compiler, and the architecture.[5] This cannot be achieved by simple extensions to conventional sequential languages and architectures—the problems of determinacy, cost of synchronization and context switches, and intolerance of memory latency are insurmountable in the pure von Neumann framework.

A. Languages and Compiling

Our research on languages is constrained mainly by the two requirements of *implicitly* parallel semantics and *determinacy*.

Originally (i.e., in [10], 1978), Id was simply intended as a convenient textual encoding of dataflow graphs which are tedious to draw explicitly. Over the years, Id has evolved in the direction of higher level features and greater abstraction. Today, the functional subset of Id is as powerful and abstract as other modern functional languages such as SML [31], LML [15], and Miranda [46]. Like these other functional languages, Id can be explained and understood purely in terms of the concept of *reduction*, without any recourse to dataflow graphs (such an explanation may be found in [13]).

I-structures and managers extend Id beyond functional languages. *I*-structures were originally introduced only as a characterization of certain "monotonic" constructions of functional arrays ([14], 1980). It was only in 1986 that the connection with logic variables became clear, and *I*-structures were clearly incorporated into the language [12]. A recent development is the "array comprehension" notation by which the programmer can stay within the functional subset and largely avoid the explicit use of *I*-structures [34]. We believe that the treatment of arrays is one of Id's unique features. Managers in Id are used for expressing explicit nondeterminism and are more expressive than the "merge" operator often used to express nondeterminism in functional languages.

It is also interesting to compile Id for sequential and parallel architectures using von Neumann processors. The major complication here arises from Id's nonstrict semantics which makes it quite difficult to achieve efficient partitioning of code into sequential von Neumann threads. This has recently been a very active area for research. Outside our group, this has been the primary focus of the *graph-reduction* [45] community, where an additional motivation has been "lazy evaluation," which is one way to achieve nonstrict semantics. Work on compiling for sequential von Neumann machines may be found in [26] and [19], and for parallel von Neumann machines in [37] and [20]. Within our group, we have recently embarked upon a project to tackle this problem systematically and at a more fundamental level [44], cleanly separating out the issue of nonstrictness from the issue of laziness. This work has, in fact, strengthened our conviction that dataflow architectures are good architectures in which to implement parallel graph reduction.

Another language associated prominently with dataflow is SISAL [30] (which, in turn, was influenced by both Id and VAL, another early dataflow language [1]). Id and SISAL differ in many ways; most notably:

- SISAL deliberately omits higher order functions in favor of simplicity.
- Current implementations of SISAL have strict semantics, although we have been informed that the SISAL specification takes no position on strictness.
- SISAL arrays are purely functional, and are extensible, i.e., the index bounds can grow. It is possible to define some arrays monolithically using the "forall" construct. In addition, there is an incremental update operation that conceptually produces a new array from an old one, differing at one index. If implemented naively, this implies some sequentialization and a heavy use of array storage, but it is the aim of SISAL researchers to use program analysis to alleviate this problem. In contrast, Id arrays are not extensible, and instead of incremental updates, the Id programmer uses bulk or mono-

[5] The RISC experience has demonstrated this tight coupling even on sequential machines.

lithic operators like map_array (the programmer can code new bulk operators himself, using higher order functions). The nonstrictness of Id is crucial in allowing bulk operators to be used when an array is defined using recurrences.

• Both Id and SISAL are statically and strongly typed. However, SISAL's type-system is monomorphic, and requires type declarations by the programmer, whereas Id has a polymorphic type system, and types are automatically inferred by the compiler.

Most current SISAL research focuses on compiling to existing multiprocessors, except at Manchester University, where the target is the Manchester dataflow machine. A major contribution of the SISAL effort has been to define IF1, an intermediate language to which SISAL programs are first translated. IF1 is a dataflow graph language, although not at a sufficiently detailed level to be a directly executable machine language. Proper documentation of IF1 and tools for manipulating it have allowed diverse research groups to target SISAL to their machines. However, none of the current implementations of SISAL can match the performance of conventional languages on parallel or sequential machines.

Lucid is another language known as a dataflow language [47] because, though textually a functional language, the operational interpretation often given to Lucid programs is one of networks of *filter* functions connected by arcs carrying infinite sequences of values. High-level iteration constructs are used to specify the filters and their interconnections. Unlike our dataflow graphs that constitute a machine language, Lucid's networks are at a much more abstract level and do not address such issues as tagging, data structure representations, etc. Current implementations of Lucid interpret such networks in von Neumann code. Insofar as Lucid can be viewed denotationally as a purely functional language, it should also be amenable to our compilation techniques. As a programming language, Lucid does not have higher order functions, arrays, user-defined data types, type checking, etc., although we understand that such features are under consideration.

B. Dataflow Architectures

The first tagged-token dynamic dataflow interpreter was the *U*-Interpreter developed in 1977 [9]. In the *U*-Interpreter, contexts and iteration numbers on tags were completely abstract entities, with no physical interpretation. Indeed, for procedure calls, each context carried within it the entire chain of its parent contexts. The TTDA is an evolution of the *U*-Interpreter in the direction of a realizable architecture. Contexts now have a physical interpretation—they are directly related to machine resources, referring to code block registers, database registers, constant areas, and so on. Detailed, explicit mechanisms have been developed to invoke new contexts and restore old ones. Loop-bounding techniques have been developed in recognition that resources are bounded (including the iteration field on a token). Finally, *I*-structure memory has been developed to deal with data structures.

Of course, the TTDA is not the end of the evolutionary path from the *U*-Interpreter. Our current view is embodied in the Monsoon dataflow processor architecture [36], which we describe briefly in the next section.

The tagged-token dataflow idea was also developed independently at Manchester University, where the first dataflow machine was built [22]. It consisted of one processing element and introduced the idea of "waiting-matching" functions. This made it possible to implement an "*I*-structure store" in the waiting-matching section itself. Although the Manchester machine was too small to run any actual applications, it was able to demonstrate that pipelines in a dataflow processor can be kept busy almost effortlessly.

The most complete and impressive dataflow machine to date is the Sigma-1, built by researchers at Japan's Electro-Technical Laboratory [23], [48]. It embodies nearly all the ideas discussed in this paper. The current implementation consists of 128 processors and 128 *I*-structure stores and has just gone into operation (early 1988). It has already demonstrated a performance of 170 MFLOPS on a small integration problem. It is programmed in dataflow C, a derivative subset of the C programming language. There is a paucity of software for Sigma-1, although we think it would be straightforward to develop an Id compiler for it.

In Japan, there is also related work at the NTT under the direction of Dr. Amamiya [3], [40]. NEC has also built some dataflow machines, including a commercial signal-processing chip [32], [42].

A more detailed survey of dataflow architectures may be found in [5].

C. Project Status and Plans: Id and the Monsoon Machine

Our current research (December 1988) continues to cover a spectrum from languages to compilers to architectures.

On the Id language, we are working on further improvements in data-structuring facilities, development of an automatic incremental, polymorphic type-checker with overloading, and language constructs for resource managers.

The central vehicle of our compiler research is the Id compiler implemented by Traub [43]. Issues we are currently investigating revolve around optimization techniques: use of type information to improve object code, code motion and transformation, fast function calls and loops, fast resource management, and reducing the overhead of dynamic resource management by moving some of those activities into in-line code.

To support this research, we have constructed *Id World*, an integrated programming environment for Id and the TTDA [33] running on workstations such as Lisp machines. In addition to sophisticated edit-compile-debug-run support, Id World measures and plots parallelism profiles, instruction counts, and other emulated TTDA machine statistics. The first version of Id World was released under license from MIT on April 15, 1987.

In our laboratory, programs can also be run on two other facilities without recompilation. The Multiprocessor Emulation Facility (MEF) is a collection of 32 TI Explorer Lisp Machines interconnected by a high-speed network that we built and has been operational since January 1986. An eventdriven simulator provides the detailed timing information essential for designing a real dataflow processor.

The Monsoon Machine: We are sufficiently encouraged by the experiments conducted to date that we are proceeding with the construction of a 256-node, 1 BIPS (billion instructions per second) machine. The architecture of this machine is called "Monsoon" and was proposed by Papadopoulos and Culler [36]. It is another evolutionary step from the TTDA (and ultimately from the *U*-Interpreter) in which the resources for a code block have a direct correspondence to "frames" or "activation records" of conventional systems. The idea is basically to allocate a "frame" of wait–match storage on each code-block invocation. This frame interpretation allows the wait–match store to be a fast, *directly addressable* memory, whereas in the TTDA it had the semantics of a potentially slow associative memory.

A context, then, is merely the pointer to a frame. Tokens now have the format $\langle S, R, v\rangle_p$ where S is a pointer to an instruction in program memory, R is a pointer to the frame, v is the datum, and p, as before, the port. The instruction now contains the offset of a location in its frame where its input tokens wait to rendezvous. When a token arrives, S is used first to fetch the instruction. The offset r encoded in the instruction, together with R, is used to interrogate exactly one location, $R + r$, in wait–match memory. If empty, this token is deposited there to wait. If full, the partner token is extracted, and the instruction is dispatched.

Interestingly, it is also possible to view R as an *I*-structure address and specify fancy *I*-structure operations using S. With minor modification to the empty/full states associated with the token-store elements, *I*-structures can be implemented on the same PE hardware.

This new architecture eliminates the CBR/DBR registers of the TTDA and thus simplifies one of its resource management problems. By combining PE's and *I*-structures it reduces the engineering effort. Most importantly, our current software will run on this machine with minimal changes in the Id compiler.

A prototype single-processor Monsoon board has been operational in our laboratory since October 1988. Single-processor boards to be plugged into workstations are expected to be available in early 1990. A 16-node multiprocessor containing Monsoon processors, *I*-structure memory, and a switching network is expected to be ready by the end of 1990.

D. Macrodataflow or Pure Dataflow?

We are often asked why we take such a "fine-grained" approach to dataflow instead of using a hybrid computation model which may be termed *macrodataflow*. Rather than adopting dataflow scheduling at the individual instruction level, one considers larger chunks or "grains" of conventional von Neumann instructions that are scheduled using a program counter, with the grains themselves being scheduled in dataflow fashion.

First, we have reason to believe that the compilation problem for macrodataflow is significantly harder. Choosing an appropriate grain size and partitioning a program into such grains is a very difficult problem [38], [24], [25], [44].

Second, the macrodataflow approach requires an ability to switch a von Neumann processor very rapidly between the threads representing different grains, and no one has yet shown convincing solutions to this problem. The Denelcor HEP [39] was one attempt at such a multithreaded architecture; however, it still had inadequate support for synchronization, with some degree of busy–waiting and a limited namespace for synchronization events.

However, the appeal of a hybrid dataflow machine cannot be denied, as it represents an evolutionary step away from a von Neumann machine. Such a "von Neumann-Dataflow" machine has been studied recently in our group by Iannucci [25]. We believe that further synthesis of the dataflow and von Neumann computation models is very likely.

E. The Future

Our main research focus is determined by our belief that declarative languages on dataflow architectures constitute the right combination for general-purpose, parallel computing. However, our experiments have given us increasing confidence that Id can be a competitive language for other multiprocessors, and that dataflow architectures can competitively support other parallel languages such as parallel Fortran or C. These are exciting alternatives which we hope will attract more research attention in the future, both within our group and without.

Acknowledgment

The ideas presented in this paper are due to the efforts of the very talented students and research staff of the Computation Structures Group of the Laboratory for Computer Science at MIT. While every member of the group is typically interested in, and contributes to several aspects of the project, we mention here some rough areas of specialization: Languages and Reduction: K. Pingali and V. Kathail; Demand-driven evaluation: Pingali and S. Heller; Compilers: Kathail originally, then K. Traub, and now J. Hicks; Architecture: D. Culler, B. Iannucci, and G. Papadopoulos; Resource Management: Culler and P. Barth; Id World: D. Morais and R. Soley; Simulator: S. Brobst, A. Chien and G. Maa; Emulator: Culler, Morais, and Traub; MEF: Papadopoulos, Soley and A. Boughton.

Our thanks to the referees for detailed comments which spurred a major reworking of this paper, and to P. Barth, S. Brobst, D. Culler, G. Papadopoulos, L. Snyder, R. Soley, and N. Tarbet for a careful reading and comments on the revision.

References

[1] W. B. Ackerman and J. B. Dennis, "VAL—A value-oriented algorithmic language: Preliminary reference manual," Tech. Rep. TR-218, Computat. Structures Group, MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, June 1979.

[2] J. Allen and K. Kennedy, "PFC: A program to convert FORTRAN to parallel form," Tech. Rep. MASC-TR82-6, Rice Univ., Houston, TX, Mar. 1982.

[3] M. Amamiya, R. Hasegawa, O. Nakamura, and H. Mikami, "A list-oriented data flow machine," in *Proc. Nat. Comput. Conf.*, AFIPS, 1982, pp. 143-151.

[4] Arvind and J. D. Brock, "Resource managers in functional programming," *J. Parallel Distrib. Comput.*, vol. 1, no. 1, Jan. 1984.

[5] Arvind and D. E. Culler, "Dataflow architectures," in *Annual Reviews in Computer Science, Vol. 1.* Palo Alto, CA: Annual Reviews Inc., 1986, pp. 225-253.

[6] ——, "Managing resources in a parallel machine," in *Fifth Generation Computer Architectures, 1986.* New York: Elsevier Science Publishers, B.V., 1986, pp. 103-121.

[7] Arvind, D. E. Culler, and G. L. Maa, "Assessing the benefits of fine-grained parallelism in dataflow programs," *Int. J. Supercomput. Appl.*, vol. 2, no. 3, Fall 1988.

[8] Arvind and K. Ekanadham, "Future scientific programming on parallel machines," *J. Parallel Distrib. Comput.*, vol. 5, no. 5, Oct. 1988.

[9] Arvind and K. Gostelow, "The U-Interpreter," *IEEE Comput. Mag.*, vol. 15, Feb. 1982.

[10] Arvind, K. Gostelow, and W. Plouffe, "An asynchronous programming language and computing machine," Tech. Rep. TR-114a, Dep. Inform. Comput. Sci., Univ. of California, Irvine, CA, Dec. 1978.

[11] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing," in *Proc. DFVLR—Conf. 1987 Parallel Processing Sci. Eng.*, BonnBad Godesberg, W. Germany, June 25-29, 1987.

[12] Arvind, R. S. Nikhil, and K. K. Pingali, "*I*-Structures: Data structures for parallel computing," in *Proc. Workshop Graph Reduction,* Santa Fe, New Mexico. Berlin, Germany: Springer-Verlag, Sept./Oct. 1986. Lecture Notes in Computer Science 279.

[13] ——, "Id Nouveau reference manual, Part II: Semantics," Tech. Rep., Computat. Structures Group, MIT Lab. Comput. Sci., 545 Technology Square, Cambridge, MA 02139, Apr. 1987.

[14] Arvind and R. E. Thomas, "*I*-structures: An efficient data type for parallel machines," Tech. Rep. TM 178, Computat. Structures Group, MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, Sept. 1980.

[15] L. Augustsson and T. Johnsson, "Lazy ML user's manual," Tech. Rep. (Preliminary Draft), Programming Methodology Group Rep., Dep. Comput. Sci., Chalmers Univ. of Technol. and Univ. of Goteborg, S-421 96 Goteborg, Sweden, Jan. 1988.

[16] D. E. Culler, "Resource management for the tagged token dataflow architecture," Tech. Rep. TR-332, Computat. Structures Group, MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, 1985.

[17] ——, "Effective dataflow execution of scientific applications," Ph.D. dissertation, Lab. Comput. Sci., Massachusetts Instit. Technol., Cambridge, MA, 02139, 1989.

[18] J. B. Dennis, "First version of a data flow procedure language," in *Proc. Programming Symp.*, G. Goos and J. Hartmanis, Eds. Paris, France, 1974. Berlin, Germany: Springer-Verlag, 1974. Lecture Notes in Computer Science 19.

[19] J. Fairbairn and S. C. Wray, "TIM: A simple abstract machine for executing supercombinators," in *Proc. 1987 Functional Programming Comput. Architecture Conf.*, Portland, OR, Sept. 1987.

[20] B. Goldberg and P. Hudak, "Alfalfa: Distributed graph reduction on a hypercube multiprocessor," Tech. Rep., Dep. Comput. Sci., Yale Univ., New Haven, CT, Nov. 1986.

[21] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, no. 2, pp. 175-189, Feb. 1983.

[22] J. R. Gurd, C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Commun. ACM,* vol. 28, no. 1, pp. 34-52, Jan. 1985.

[23] K. Hiraki, S. Sekiguchi, and T. Shimada, "System architecture of a dataflow supercomputer," Tech. Rep., Comput. Syst. Division, Electrochemical Lab., 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[24] P. Hudak and B. Goldberg, "Serial combinators: "Optimal" grains of parallelism," in *Proc. Functional Programming Languages Architures.* Nancy, France, pp. 382-399. Berlin, Germany: Springer-Verlag, Sept. 1985, Lecture Notes in Computer Science 201.

[25] R. A. Iannucci, "A dataflow/von Neumann hybrid architecture," Ph.D. dissertation, Lab. for Comput. Sci., Massachusetts Institute of Technology, Cambridge, MA 02139, May 1988.

[26] T. Johnsson, "Efficient compilation of lazy evaluation," *ACM SIGPLAN Notices,* vol. 19, no. 6, pp. 58-69, June 1984. *Proc. ACM SIGPLAN '84 Symp. Compiler Construction.*

[27] ——, "Lambda lifting: Transforming programs to recursive equations," in *Proc. Functional Programming Languages and Comput. Architecture,* Nancy, France. Berlin, Germany: Springer-Verlag, Sept. 1985. Lecture Notes in Computer Science 201.

[28] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proc. 8th ACM Symp. Principles Programming Languages,* Jan. 1981, pp. 207-218.

[29] D. J. Kuck, D. Lawrie, R. Cytron, A. Sameh, and D. Gajski, "The architecture and programming of the Cedar System," Tech. Rep. Cedar No. 21, Lab. for Advanced Supercomput., Dep. Comput. Sci., Univ. of Illinois at Urbana-Champaign, Aug. 12, 1983.

[30] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, P. Hohensee, and I. Dobes, "Sisal reference manual," Tech. Rep., Lawrence Livermore Nat. Lab., 1984.

[31] R. Milner, "A proposal for standard ML," in *Proc. 1984 ACM Symp. Lisp Functional Programming,* Aug. 1984, pp. 184-197.

[32] NEC, *Advanced Product Information User's Manual: μPD7281,* NEC Electronics Inc., Mountain View, CA, 1985.

[33] R. S. Nikhil, "Id World reference manual," Tech. Rep., Computat. Structures Group, MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, Apr. 1987.

[34] ——, "Id (Version 88.1) reference manual," Tech. Rep. CSG Memo 284, MIT Lab. for Comput. Sci., 545 Technology Square, Cambridge, MA 02139, Aug. 1988.

[35] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM,* vol. 29, no. 12, Dec. 1986.

[36] G. M. Papadopoulos, "Implementation of a general-purpose dataflow multiprocessor," Ph.D. dissertation, Lab. for Comput. Sci., Massachusetts Instit. Technol., Cambridge, MA 02139, Aug. 1988.

[37] S. L. Peyton-Jones, C. Clack, J. Salkild, and M. Hardie, "GRIP—A high performance architecture for parallel graph reduction," in *Proc. 3rd Int. Conf. Functional Programming Comput. Architecture,* Portland, OR, Sept. 1987.

[38] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow," in *Proc. 1986 ACM Conf. Lisp Functional Programming,* Cambridge, MA, Aug. 4-6, 1986, pp. 202-211.

[39] B. J. Smith, "A pipelined, shared resource MIMD computer," in *Proc. 1978 Int. Conf. Parallel Processing,* 1978, pp. 6-8.

[40] N. Takahashi and M. Amamiya, "A dataflow processor array system: Design and analysis," in *Proc. 10th Int. Symp. Comput. Architecture,* Stockholm, Sweden, June 1983, pp. 243-250.

[41] P. Tang, C.-Q. Zhu, and P.-C. Yew, "An implementation of Cedar synchronization primitives," Tech. Rep. Cedar No. 32, Lab. for Advanced Supercomput., Dep. Comput. Sci., Univ. of Illinois at Urbana-Champaign, Apr. 3, 1984.

[42] T. Temma, S. Hasegawa, and S. Hanaki, "Dataflow processor for image processing," in *Proc. 11th Int. Symp. Mini and Microcomputers,* Monterey, CA, pp. 52-56, 1980.

[43] K. R. Traub, "A compiler for the MIT tagged token dataflow architecture," Master's thesis, Tech. Rep. TR-370, MIT Lab. for Comput. Sci., Cambridge, MA 02139, Aug. 1986.

[44] ——, "Sequential implementation of non-strict languages," PhD dissertation, MIT Lab. for Computer Sci., 545 Technology Square, Cambridge, MA 02139, May 1988.

[45] D. A. Turner, "A new implementation technique for applicative languages," *Software: Practice and Experience,* vol. 9, no. 1, pp. 31-49, 1979.

[46] ——, "Miranda, A non-strict functional language with polymorphic types," in *Proc. Functional Programming Languages Comput. Architecture,* Nancy, France. Berlin, Germany: Springer-Verlag, Sept. 1985. Lecture Notes in Computer Science 201, pp. 1-16.

[47] W. W. Wadge and E. A. Ashcroft, *Lucid, The Dataflow Programming Language.* London, England: Academic, 1985.

[48] T. Yuba, T. Shimada, K. Hiraki, and H. Kashiwagi, "Sigma-1: A dataflow computer for scientific computation," Tech. Rep., Electrotechnical Lab., 1-1-4 Umesono, Sakuramura, Niiharigun, Ibaraki 305, Japan, 1984.

Arvind (SM'85), photograph and biography not available at the time of publication.

Rishiyur S. Nikhil (M'87), photograph and biography not available at the time of publication.