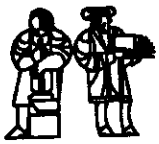
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Implicit Serialization
in Dataflow Programs:
Input/Output in Id

Computation Structures Group Memo 277
December 15, 1987

Richard Mark Soley

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Implicit Serialization in Dataflow Programs: Input/Output in Id

Richard Mark Soley

December 15, 1987

Abstract

The Id computer language, the primary input language for coding the M.I.T. Tagged-Token Dataflow Architecture and other related architectures, is well developed and relatively widely used in research settings. Id's expressive power (based on its mostly functional nature) has led to the translation of large programs into Id's compact and understandable representations.[3] In addition, research in harnessing the great amounts of parallelism immediately obvious in Id programs has been fruitful.[2] Nevertheless, a great gap in the utility of the language remains.

All new users of Id are astounded at Id's complete lack of input/output facilities. There is no support for file, terminal, printer or any other interaction mechanisms, despite the fact that such mechanisms exist within the systems used to implement the Tagged-Token architecture. This lack is due entirely to the mode of *serialization* used to compile Id programs; in order to expose maximum parallelism, sequential execution is limited to portions of programs containing data dependencies.

This paper presents an automatic scheme for the serialization of input/output within Id programs; the results are apparent to Id programmers, but no explicit notation is needed to achieve correct ordering of I/O in programs.

1 Introduction

One of the unique features of the Id language is that program synchronization, like the exploitation of program parallelism, is determined entirely by the compiler and run-time system, without the intervention of the programmer.[1] Although synchronization is a trivial problem in the general sequential-machine case (each instruction must wait for the termination of the linearly previous instruction), the uncovering of potentially great amounts of parallelism in a program invokes a concomitant need to synchronize the far-flung portions of the execution tree (*i.e.*, to "join" forked execution branches). This need arises in any area that requires concurrent access to resources, including (1) access to program memory; (2) access to data memory (I-structures[4]); (3) access to streams[6], which might be used to synchronize execution; and (4) access to input/output devices.

Solutions for two of these synchronization problems under a Tagged-Token Dataflow paradigm exist. These include the following:

- Concurrent program memory access is managed at compile time by explicitly joining execution tree branches (static links in the dataflow graph). The waiting-matching section of the machine completes the "join" operation at run time.
- The I-structure controller portion of the Tagged-Token machine implicitly handles all I-structure synchronization (dynamic links in the dataflow graph) at run time, maintaining atomicity of memory operations and completing dynamic links in the dataflow graph.

However, no complete, convenient model for either stream computation, efficient distributed manager calls, nor I/O has been developed. This paper suggests a simple, elegant approach to solving these problems.

1.1 The Input/Output Problem

In particular, an immediate solution for I/O is desirable, and might point the way to a more general serialization paradigm. The basic problem of I/O in the Id language[5] is that no data dependencies exist between successive calls to input or output functions. Since any Id compiler must rely on such "obvious" dependencies to generate the dataflow graph for execution, the run-time ordering of several I/O calls might be incorrect. The following two examples clarify this problem.

The code

```
def output x = { call write x; call write (x + 1) };
```

is clearly intended to simply write out the values of the variable x, and $x + 1$. However, as there is no dependency between the arguments to the two calls to *write*, the calls may be executed in parallel, perhaps causing $x + 1$ to be printed before x. Likewise, the code

```
def prompt_and_read prompt stream =
    { call write prompt stream in read stream };
```

is intended to prompt the user with some question, and then read and return an answer. However, again there are no dependencies between the two function calls in the program. Therefore, the *read* might execute before (or *during!*) the execution of *write*. Clearly this is unacceptable. This problem has kept all Id codes to date from containing any meaningful I/O usage.

In fact, programmers could inform the compiler of the need to perform I/O operations in some particular order by explicitly outlining the required ordering with "fake" data dependencies. For example, we could rewrite our first program above in this form:

```
def output x =
    { dependency = write x 0 in
      write (x + 1) dependency };
```

The first call to *write* would begin as soon as the argument x to the *output* function was available; however, the second call to *write* would be "detained" until the first call returned a value†, due to the link created by the variable named *dependency*. This solves the problem; using this simple technology of "dummy" ordering variables, we can inform the compiler of any particular ordering we wish to impose on I/O function calls.

However, this simple approach is actually too simple minded. Using this methodology, the simple Id program

```
def example x y =
    { call write "The square root of ";
      call write x;
      call write " is ";
      call write (sqrt x);
      total = 0
      in
         { for z from x to y do
              call foo z;
              next total = total + bar z
           finally write total
         }
    };
```

takes on the horrendous and unreadable form

†We assume that the writer of *output* would make sure that no value is returned until the actual writing is complete.

```
def example x y =
    { dep0 = write "The square root of " 0;
      dep1 = write x dep0;
      dep2 = write " is " dep1;
      dep3 = write (sqrt x) dep2;
      dep4 = 0;
      total = 0;
      temp = 0
      in
         { for z from x to y do
               next dep4 = foo z dep3;
               next temp, next dep3 = bar z (next dep4)
           finally write total dep3
         }
    };
```

Needless to say, this is a terrible state of affairs, since simple programs become unreadable, and quite difficult to write. In particular, the programmer must create many dummy variables to state dependencies (the *dep* variables above); one is necessary for *each statement in the program which potentially performs I/O.* For I/O occurring within loops (such as the *for* loop above), *"nextified"* dummies must be used to circulate the "thread of serialization." Also, it is important to note that every function call which *potentially performs I/O* must be treated in this highly confusing manner. In particular, although it isn't outlined above, the *example* routine itself probably should take an extra argument (and return an extra value) to fit into the serialization thread of its caller.

2 The Compiler Can Perform Implicit Serialization

Generally, programmers are used to an implicit "dependency thread" between calls to I/O functions based on the explicit ordering of all statements in the code. Needless to say, it is quite simple for a compiler for a dataflow language to take some account of the explicit ordering of I/O function calls when constructing the dataflow graph. If all programs in the language were composed of only one monolithic function, we might simply impose an "invisible dependency thread" on the ordering of I/O function calls. For example, in the code

```
def monolithic_code x =
    { call write "This is the value of X: ";
      call write x
    };
```

our compiler could impose a dependency (not evident from the data flow of the program) between the two *write* calls. In particular, we might want to insure that the second call to *write* should not start until the first call to *write* is complete.

In the presence of multiple unrelated I/O channels, we might want to loosen this hold partially by only enforcing these dependency threads through calls to "related" channels. For example, in the code

```
def two_channel_code x file1 file2 =
    { channel1 = open file1;              ! 1
      channel2 = open file2;              ! 2

      call write "X is: " channel1;       ! 1
      call write "X is: " channel2;       ! 2
      call write x channel1;              ! 1
      call write x channel2;              ! 2

      call close channel1;                ! 1
      call close channel2                 ! 2
    };
```

we would like to enforce a thread of dependency through the statements labelled with a 1 and a *separate* of dependency through the statements labelled with a 2. Thus we get the time ordering of execution needed for each channel, but we do not restrain the parallelism of the program unduly. Of course, if *channel*1 and *channel*2 represented two claims on the same physical I/O resource (for instance, if they represented the input and output sides of a single I/O channel to a user's terminal) we would want to thread all calls in the above program together.

Unfortunately, this does not solve the whole problem. No readable, useful codes contain only a single monolithic procedure; calls to subsidiary functions, which might invoke calls on I/O resources, are quite likely. In a broad sense, we therefore need to thread a serialization thread through (1) all calls to I/O functions and (2) all calls to any other function that might potentially cause I/O. In general, however, for any particular function application we cannot at compile time tell whether the function being called might perform any I/O. We need a way to serialize the I/O within the function called (the "callee") with the I/O in the calling function (the "caller") *without serializing the caller in toto*.

2.1 Solution 1: The I/O Trigger

A straightforward solution to this problem would be to create the dependency thread outlined above for each procedure between all I/O calls and all other calls as well. However, instead of insisting that non-I/O calls "complete" (*i.e.*, terminate) before any further calls be made, we need only insist that any *I/O in the callee completes* before any new I/O is initiated by the caller.

To introduce this dependency between procedures, we propose that every function have an extra "hidden" argument, in addition to the declared arguments, which is passed in the usual manner. We further propose that every function have an extra "hidden" result, in addition to the result arity of the procedure. The extra incoming argument should be the origination ("trigger") of the dependency thread for the body of that procedure; likewise, the extra result of the function would be generated by the output of the dependency thread of that function.

Wherever an I/O procedure is called, the compiler threads the I/O serialization thread such that the I/O procedure may not *start* until triggered by the dependency thread. The compiler also continues the dependency thread as an output of the I/O procedure.

In addition, when a non-I/O procedure is called, the compiler threads the I/O serialization thread as the extra argument to that function. The thread continues as the extra result of the procedure.

This alteration of the dataflow graph is outlined graphically in Figure 1. The procedures graphed correspond to the following code:

```
def show_squares n =
    { call write "The square root of ";
      call write n;

      input = check n;
      answer = sqrt input;

      call write " is ";
      call write answer
    };

def check number =
    if number < 0
      then { call write " [absolute]" in − number }
      else number;
```

The dashed lines indicate the new dependency threads that would not be placed by a compiler that relied only on data dependencies. Note that the compiler, knowing that the primitive function *sqrt* will never cause any I/O activity, might forgo the inclusion of the *sqrt* call in the dependency thread.

This solution does not give us enough power, however. We might wish to support multiple concurrent access to separate, unrelated I/O channels. The procedure outlined above would over-restrain the parallelism of such a program, as it would serialize all I/O occurring within the program.

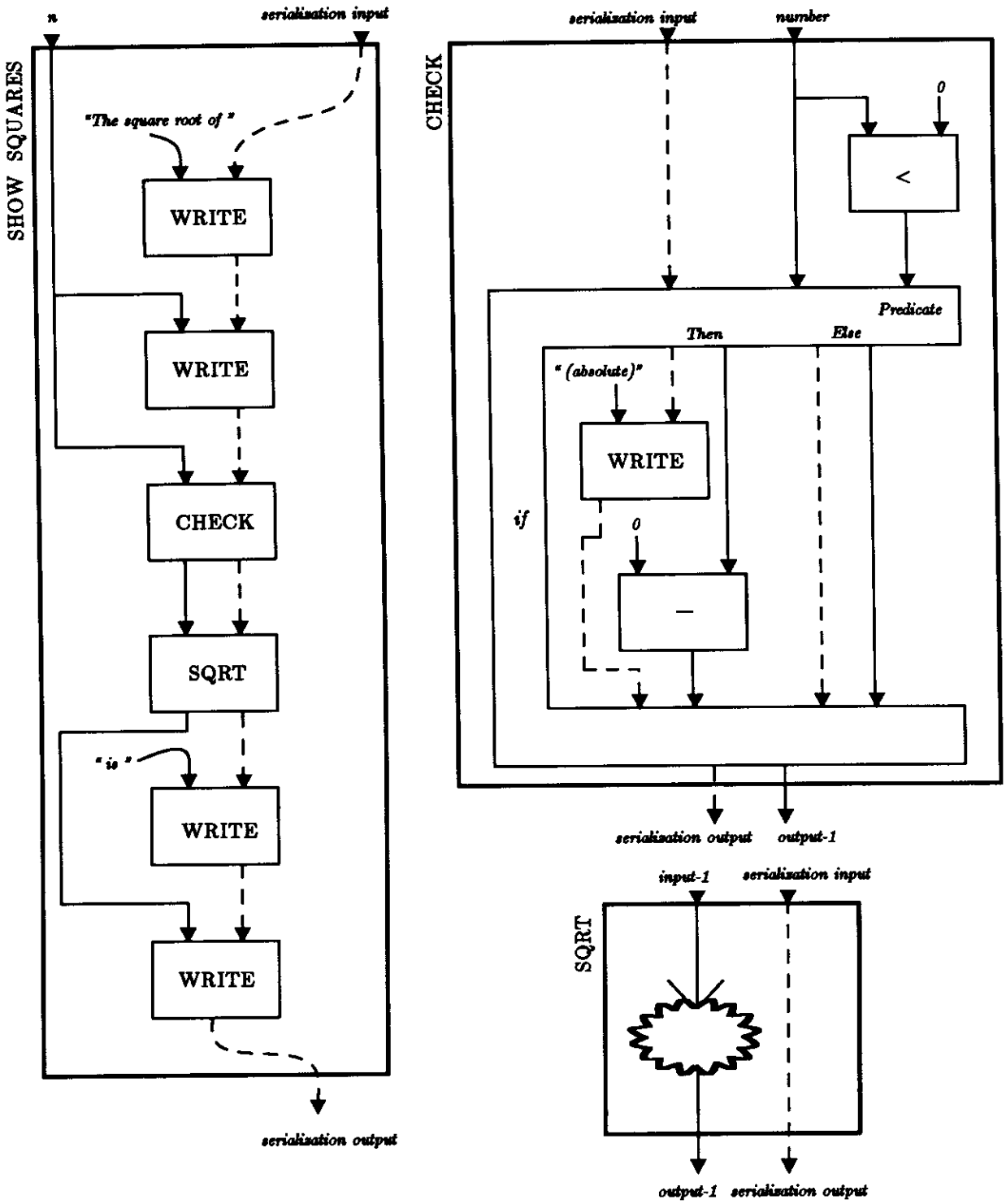


Figure 1: Show_Squares and Related Procedures with I/O Triggers

2.2 Solution 2: Multiple I/O Triggers

To alleviate this problem, we could expand the schemata to allow any number of extra serialization trigger arguments and results to be added. If we wanted to support concurrent access to n channels, we would then need to expand the number of extra arguments and results to n to allow the threading of n different serialization threads through functions. An example of this scheme is outlined in Figure 2, which is the (simplified) graph for the following code:

```
def two_channel_code x file1 file2 =
    { channel1 = open file1;
      channel2 = open file2;
      call root x channel1;
      call root x channel2;

      call close channel1;
      call close channel2
    };

def root x channel =
    { call write "The answer is: " channel;
      call write (sqrt x) channel)
    };
```

However, this solution has two major drawbacks. First, it requires a compile-time static declaration of the number of channels which will be used during computation. Second, it also requires that each and every function and primitive used must be recompiled to take the extra arguments and return the extra values, even if the function makes no claim on *any* I/O resources. This is due to the fact that the compiler cannot always make a compile-time decision on whether a function might perform any I/O. Worse, even if the compiler *could* make such a determination, codes such as

```
(if pred arg then func_1 else func_2) arg1 arg2 arg3
```

could not be efficiently compiled to take advantage of the knowledge. Arbitrary complexity might need to be added to otherwise simple dataflow graphs to support this scheme. Figure 2 doesn't show the whole picture; in fact, each and every procedure call must be passed (and return) *every* serialization thread. The complex nature caused by this constraint is clear in Figure 3, showing all of the necessary serialization thread switching and choices.

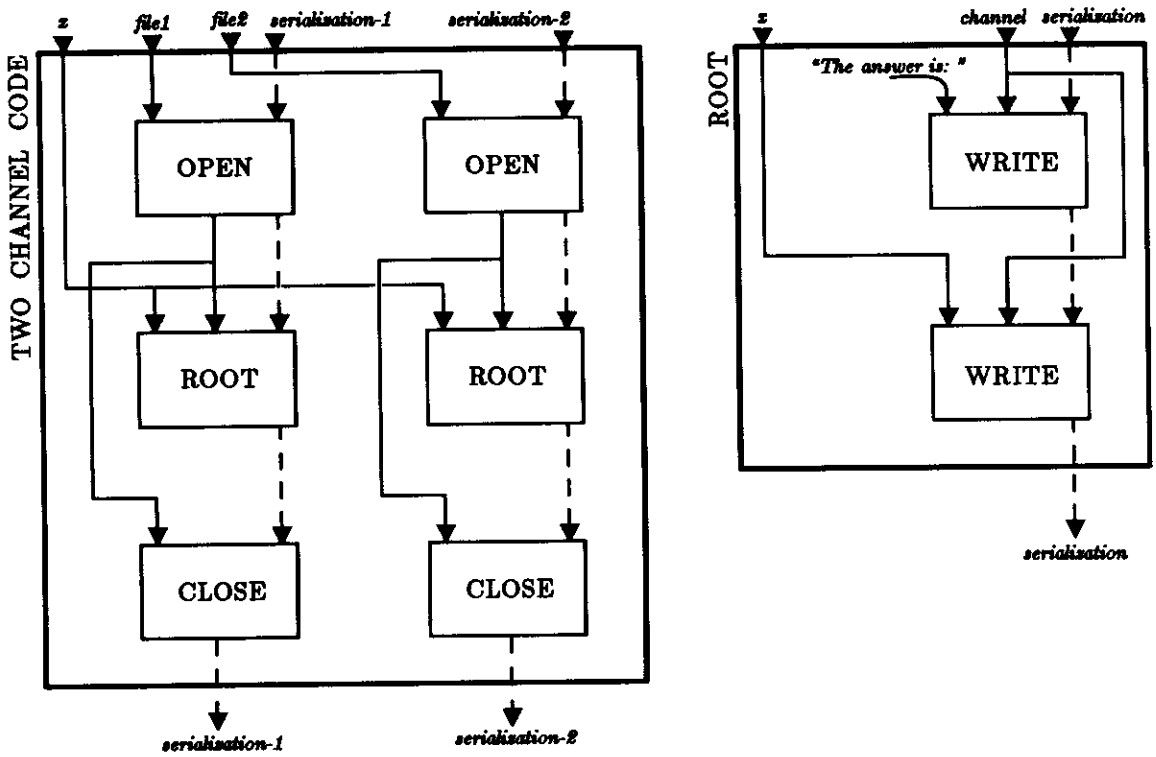


Figure 2: Two_Channel_Code Related Procedures with Multiple Triggers

2.3 Solution 3: Trigger Structures

Using the only data structuring ability of the Tagged-Token machine (I-structures) comes immediately to mind to solve this problem. Indeed, I-structures provide the most elegant conceptual solution. As in Solution 1, we enforce the regime of an extra argument and an extra return value for every function ever compiled. However, instead of representing the dynamic arc of a single serialization thread, this argument (and return value) would comprise an I-structure of all dynamically current serialization threads. Since the size of the structure can be determined dynamically, a single static compilation of all procedures would be sufficient.

Under this paradigm, procedures that make no claim on any I/O resources would simply pass the incoming serialization vector as the immediate serialization vector result. Procedures which do make such claims would "thread into" particular slots in the serialization vector. For example, the code in the previous example would compile into the graphs displayed in Figure 4.

Note that the *open* function is now expected to simply choose (and return) an offset into the serialization vector. The I/O calls such as *write* must then serialize the primitive I/O requests that they represent, as in Figure 5. In Figure 5, the code represented by *CopyAllBut* corresponds to the following I-structure exclusive copying procedure, with *old*, n_1, n_2, ... n_n as inputs:

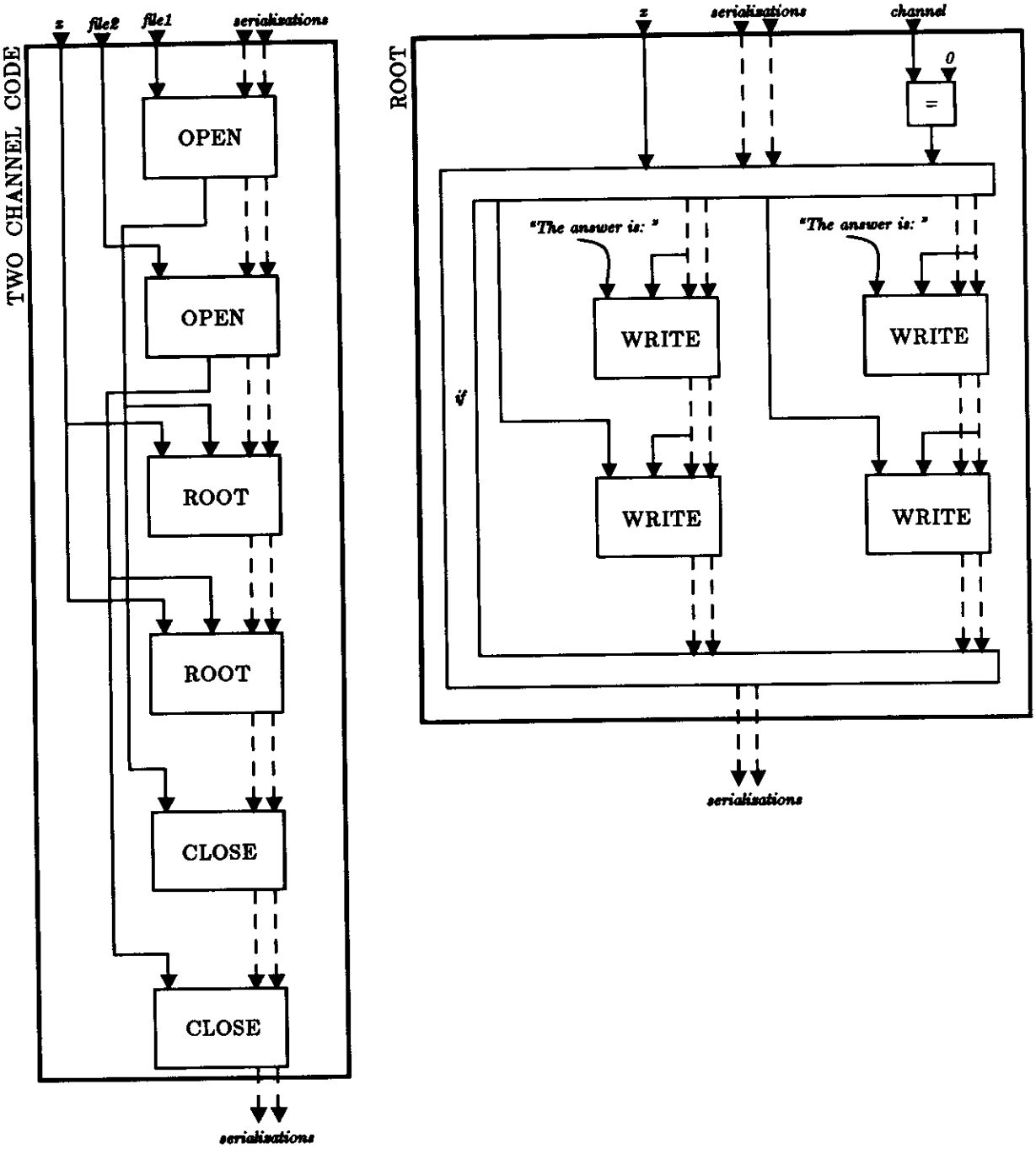


Figure 3: Two_Channel_Code with Multiple Triggers and All Serialization

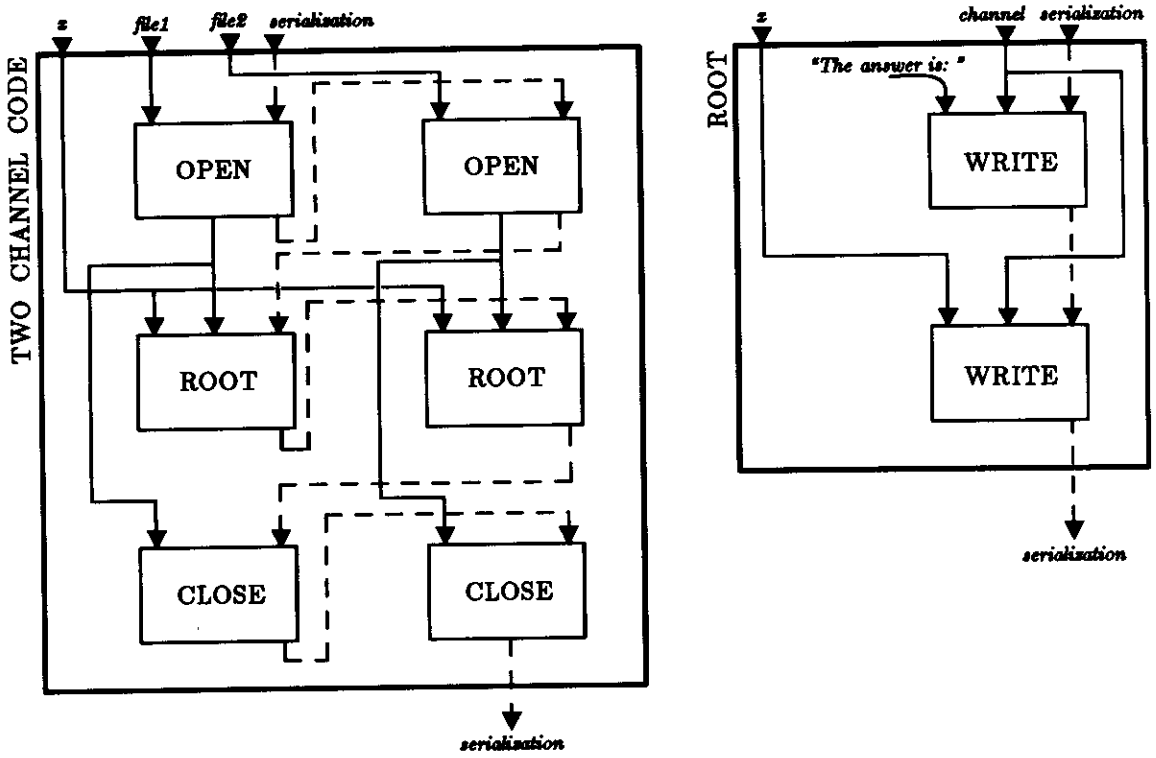


Figure 4: Two_Channel_Code Related Procedures with Trigger Structures

```
{ low, high = bounds old
  new = array (low, high) in
  { for i from low to high
        if (i <> n1) and (i <> n2) and ... and (i <> nn)
          then new[i] = old[i]
        finally new
  }
}
```

This procedure copies an I-structure completely, except for any elements of the structure that represent serialization threads used by the local procedure. *PrimitiveWrite* represents the lowest level instruction for initiating an output request, and waiting for completion of the output operation.

The most unfortunate side-effect of this approach is the terrific expenditure in I-structure operations. There is no I-structure overhead for functions which have no I/O side effects, but the I-structure copying necessary for I/O performing functions could be quite large.

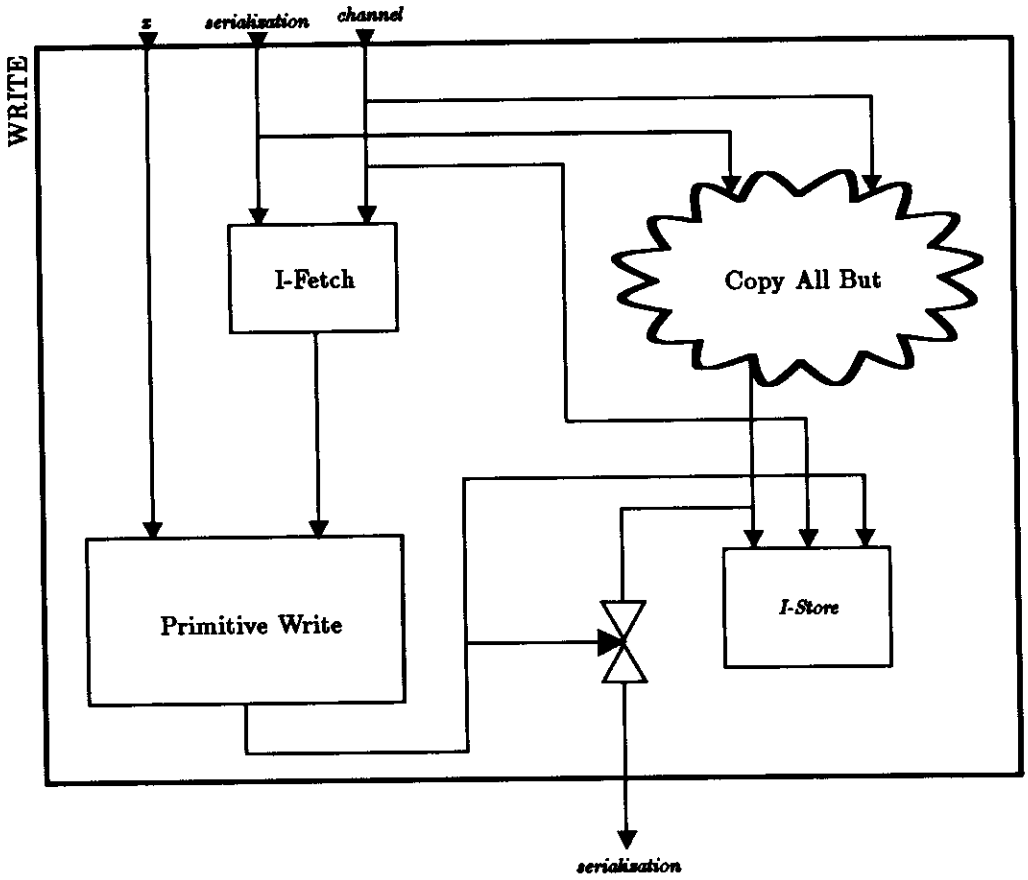


Figure 5: Serialization of Primitive Writes in the Write Procedure

3 An Optimization for Queueing I/O

To this point in our discussion we have assumed that the primitive I/O performing procedures could be capable of passing on their serialization trigger outputs at some point after the I/O request was actually complete. This has two problems:

- We might not be able to support these semantics on some real machine, or the expense of such semantics might be prohibitive, depending on the exact implementation of I/O request handling.
- These semantics preclude the overlapping of I/O where it is possible. In particular, we don't mind the misordered *queueing* of I/O requests, only the misordered *execution* of those requests.

Note also that to this point in the discussion we have not used the data field of the serialization tokens for any useful purpose, instead treating the token merely as a trigger for starting computation. An interesting optimization (which may be applied to any of the above solutions) solves the strict I/O ordering problem, taking advantage of this unused field.

In the modified schemata, the data field of serialization tokens becomes a *serialization sequence number*. Non-I/O operations that take serialization tokens as input, producing an output serialization token, would output a token with a sequence number identical to the input trigger. I/O functions, however, would increment this sequence number by one, and queue their I/O request *with the sequence number*. Thus, it is possible to misorder I/O requests, relaxing the strict I/O precedence constraints outlined above, without sacrificing the strict ordering of outputs.

The *manager* handling the actual I/O requests for a particular channel is then responsible for handling the requests in a strictly sequential order, causing the I/O functions to be issued in the order that the programmer intended, *rather than the order in which the requests happened to be issued.*

As is hinted by the terminology used above, it is believed that this approach to synchronization might be applied to streams other than I/O channels. Since the latter solutions allow us to generate any number of statically named streams, we could use this device to implement other streams of data.

4 Implementation of I/O Serialization for Id

An implementation of I/O serialization within the compiler for the Id language has been completed. In this implementation, there is a single static serialization thread through each compiled procedure, and dynamic linkage is provided at function application time to connect the serialization thread of the caller and the callee. Tokens traveling this thread contain an I-structure descriptor, which specifies an I-structure with as many slots as the maximum number of I/O channels in use (as in solution three specified above). Each slot can contain the following:

- *NIL* specifies that the channel number is not in use, and may be allocated by a call to the *open* procedure.
- An *empty* slot denotes that a channel number is in use, and I/O is currently pending on that channel (i.e., later I/O procedures using that channel will wait).
- An *IOD*, or I/O descriptor, specifying a named stream controlled by the input/output unit of a particular processing element.

4.1 Compiler Modifications

The Id compiler was changed in several ways to make static serialization threading possible. Since the extant compiler never needed to note lexographic ordering in any compilation phase, this information used to be discarded after the syntax error checking phase. The first change to the compiler involved maintaining lexographic ordering information for all function applications, and any block encapsulating structures, such as conditionals and loops. These structures, which in the Id compiler's program graph representation are all treated as individual "instructions," are then sorted for static serialization.

To clarify this sorting process, we introduce the concept of *strict encapsulation.* The Id compiler makes note of all program graph instructions within a procedure definition block which themselves contain basic blocks.[7] These "macro" instructions are said to *encapsulate* the instructions making up the contained basic blocks. We say that such instructions are *strictly encapsulated* by the encapsulating structure when no intermediate encapsulators are present. For example, in the following program

```
def foo a x =
    { for y from 1 to x do
          a[y] = if odd? y
                 then 0
                 else y + 1
    };
```

the + instruction is encapsulated by both the *for* and *if* structures, but only *strictly* encapsulated by the *if* structure.

Within each basic block, all strictly encapsulated operators are sorted by strict lexographic ordering, left-to-right and top-down. This ordering is then taken as the implied static ordering of I/O operations in that block. The recursive algorithm is outlined below; this procedure is called with a single argument, the *definition* instruction node representing an Id program.

```
define static_serialization_thread (node)
        for each subnode in strictly_encapsulated_nodes_in (node)
            if encapsulator? (node)
              collect static_serialization_thread (subnode)
            else if application? node collect subnode
        finally sort collected nodes lexigraphically
```

There is a potential danger in this particular implicit serialization scheme which becomes apparent in the lexographic sorting phase. A possible deadlock can be introduced, especially by the Id construct $f(gx)$, which specifies that the function f should be applied to the result of applying the function g to x. As f depends on the result of g to compute a result, there is a data dependency from the application of g to the application of f. However, since f appears lexographically before g in the program, there will be a serialization dependency from f to g. This is not guaranteed to cause a deadlock, but it makes such dangers slightly more likely. This problem might have to be dealt with by some future scheme.

4.2 TTDA Modifications

In order to try out the modified Id language with implicit I/O ordering, changes to the Gita Tagged-Token dataflow machine interpreter were carried out to support serialization. These modifications included changes to match the compiler's new output for function definition and application. Since the serialization scheme outlined in this paper required an additional entry value into and exit value from each function definition, Gita was altered to recognize the new entrypoint and return value. The only changes required were to the mechanism for invoking procedures, and the initialization "boot code block" which receives return values from top-level function calls.

Many more important changes to Gita were required to support I/O itself, as opposed to general serialization. A new, per-processing element operational unit was added to the machine simulation. This "I/O unit" operates much like an I-structure storage unit.[4] It defers read requests until input is available, and acknowledges write requests upon completion. Like I-structure operations, I/O operations are triggered by standar TTDA instructions which send *I/O Request Packets* to I/O units to open, close, read and write. Open streams are represented by *I/O descriptors* which specify the processing element whose I/O unit is handling the stream, and a local index of the stream (i.e., name of the stream local to the controlling I/O unit). These I/O descriptors are stored in the slots of serialization token I-structures shuttled along the serialization threads outlined in this paper.

5 Acknowledgments

The author wishes to thank Kenneth Traub for helping to work through the ideas in this paper, and for his marvelous compiler technology which made the implementation task bearable. Mr. Traub and Steve Heller provided enthusiastic support for the ideas. Isabel Szabó provided continuous support and conversion from technobabble into English.

References

[1] Arvind and David E. Culler. Dataflow architectures. In *Annual Review of Computer Science, Volume 1*, pages 225–253, Annual Reviews, Inc., Palo Alto, California, 1986.

[2] Arvind and David E. Culler. Managing resources in a parallel machine. In *Fifth Generation Computer Architectures*, pages 103–121, Elsevier Science Publishers B.V., 1986.

[3] Arvind and K. Ekanadham. *Future Scientific Programming on Parallel Machines.* Computation Structures Group Memo 272, M.I.T. Laboratory for Computer Science, 1987.

[4] Steven K. Heller. *An I-Structure Memory Controller.* Master's thesis, Massachusetts Institute of Technology, 1983.

[5] R. S. Nikhil and Arvind. *Id Nouveau.* Computation Structures Group Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[6] Keshav Pingali and Arvind. Efficient demand-driven evaluation, Part I. *ACM TOPLAS*, 7(2):311–333, 1985.

[7] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* Master's thesis, Massachusetts Institute of Technology, 1986.