JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

Implementing a Functional Language on the RMIT Dataflow Architecture

TR 112 074 R

Neil J. Webb †‡
Paul G. Whiting †
Robert S.V. Pascoe ‡

† Division of Information Technology
C.S.I.R.O
‡ c/o Departments of Computer Science and Communication and Electronic Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001
Australia

Version 1.0 September 1988

ABSTRACT:

This paper introduces a variant of a dataflow machine which is under development at the Royal Melbourne Institute of Technology, and discusses some issues associated with an implementation of a variant of the Id Nouveau programming language (IDA) for this environment.

1. INTRODUCTION

Traditionally, programming languages have been executed on sequential architectures with a total ordering of operations specified by the programmer. In these languages, where requirements for computational speed or other considerations dictate that segments of a program be executed concurrently, it has been the programmer's responsibility to specify both the task-wise partitioning of the code, and to take total responsibility for the start-up, communication/synchronisation and close-down of these tasks.

Under the dataflow model of computation, any operation may be performed when its data becomes available. A pure dataflow program is a directed graph, where the nodes represent operations, and the arcs represent values. This approach is functional and, in common with more traditional functional programming, has the attribute that it has no implicit temporal dependencies; outputs from operations depend only on the arriving arguments, and not on other execution history or state-components.

This attribute is significant from two aspects, one methodological, and associated with the subject of this paper:

(i) Providing the underlying architecture is also of a dataflow nature (i.e. a network of processors which *fire* when their data arrives), a massively parallel execution environment is readily produced, and - more importantly - this concurrency requires no commitment from the programmer.

(ii) Optimisation is simplified as there is no concept of state. For example, common sub-expression elimination is trivial as expressions, within a certain name-space, represent identical values compared with traditional languages where textually identical sub-expressions may depend on variables which have changed in value. Other optimisations include loop unraveling, code motion, code inlining and dead code elimination.

The issue is complicated (in common with other functional programming approaches), as the underlying architecture is far from *pure*. This is due to a number of pragmatic considerations needed to increase efficiency and render the pure functional form practical.

1.1. Project

The Joint RMIT / CSIRO Parallel Systems Architecture project officially commenced in May 1986. It is a joint collaborative project between the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organisation. The purpose of the project is to investigate parallel algorithms, methodologies, languages and architectures. Given the resources, the project has concentrated on the dataflow model of computation [10].

1.2. Hardware

The variant of dataflow being used is based on an architecture which was first proposed in 1976 by Egan at Manchester University, UK [7, 8] and subsequently developed at RMIT [9]. A multiprocessor emulation facility is available for high speed interpretation of the programs as well as a conventional discrete-event simulation of the architecture. Work is currently progressing on the design of faster processing elements which will provide a high speed multiprocessor computer.

1.3. Software

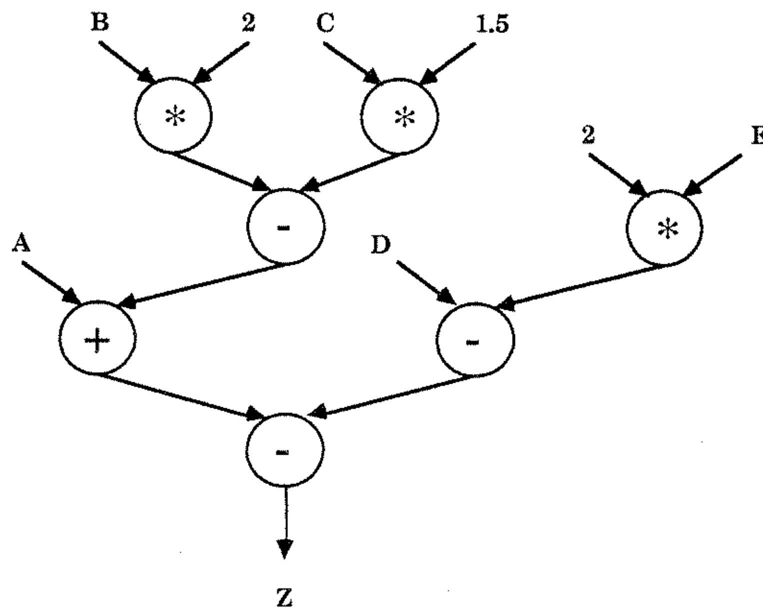
Compilers for a number of languages are under development. These include IDA (a version of the near functional language Id Nouveau [11]), GHC (Guarded Horn Clauses, a logic programming variant with explicit parallelism and committed choice [12]), SISAL (Streams and Iteration in a Single Assignment Language [13]) and Pascal - a standard version exploiting the possible concurrency available in Pascal programs. DL1 (Dataflow Language 1 [14]) was completed as a local language in the early stages of the project.

2. DATAFLOW MODEL

A dataflow program is represented by a directed graph. The nodes represent machine instructions and the arcs represent values in transition between instructions (See figure 1). The complexity of the instructions varies widely - from a simple addition to the extraction of a sub-vector from an array. Values (tokens) are placed on and removed from arcs according to firing rules.

The underlying machine consists of a set of processors, each of which may support one or more nodes in the graph, and a communications network.

The primary aim of a parallel computer is the achievement of increased processing speed, and subsequently, reduced execution times. The maximum degree of concurrency is equivalent to the breadth of the unraveled dataflow graph. Theoretically, this means that the total time for the program to execute is only as long as the time for the critical path of the program to be computed.




```
Z = A + (B*2 - C*1.5) - (D - 2*E)
```

Figure 1. - Example Dataflow Graph

3. THE LANGUAGES

A number of choices need to be made in the implementation of any dataflow language. The first of these is the choice between demand-driven and data-driven execution. The demand-driven model operates on the requirement of a node for its data. This is achieved by a node sending a trigger signal back to each preceding node which directly

produces its arguments. These signals propagate back until arguments are either literal or are input data, and the forward execution sequence begins. While this lazy evaluation model prevents unwanted results from being computed (for example the unused arm of a conditional expression), it serialises the computation, and places extra demand on the communication channel because of the back-triggers.

The alternative to demand-driven execution is data-driven. In this approach, nodes execute when the data arrives. When all the arguments to a node are available, it can fire. Firing consumes the arguments, performs the specified operation, and places the result on its output.This eager-evaluation approach maximises parallelism, and places no extra demand on the communication network since the trigger is the arrival of the data itself. Eager evaluation can however lead to an infinite chain of execution, for example, the partial generation of sub-expressions in a recursive function.

There is another implementation issue. How are the nodes on the graph distributed across physical processors? Attempting to clump nodes together in the one processor based on their locality in the graph aims to minimise communication cost. Unfortunately, evidence so far suggests that this scheme creates execution *hot spots*, and a random static allocation scheme combined with random dynamic allocation for re-entrant or recursive calls, is currently being used at RMIT. This allocation scheme has been found to perform well for programs of non-trivial size.

3.1. Introduction

There are two parallel functional languages that were considered for implementation, SISAL (Streams and Iteration in a Single Assignment Language) and Id Nouveau. SISAL was developed as a cooperative effort between Colorado State University, Digital Equipment Corporation (DEC), Lawrence Livermore National Laboratory and the University of Manchester, and is part of an international effort to evaluate different architectures for future parallel machines. Id Nouveau was developed at Massachusetts Institute of Technology and is used on their tagged-token dataflow architecture (TTDA)[20].

After evaluation, it was decided to implement both languages. The implementation of Id Nouveau (IDA) provides the ability to compare MIT's TTDA and Manchester's dataflow architecture with RMIT's hybrid dataflow machine. Implementing SISAL allows comparisons between other SISAL compilers for a host of computer architectures ranging from Crays, WARPs and Transputers to Sequents, Encores, Suns and Vaxen. This then provides a direct link, with identical source code, for benchmarking against the other architectures and the RMIT dataflow machine.

3.2. IDA

Id Nouveau is MIT's experimental dataflow language. It is a functional language with powerful constructs designed to attain optimal concurrency from existing and novel algorithms.

3.2.1. Language Description

IDA is a collection of named, value returning functions. Each function may contain a series of bindings to local variables, and these execute in parallel as the values of their generating expressions become available. To preserve the functionality of the language, each variable may be bound only once. In the following example (figure 2), the three assignment statements are constrained by data dependencies so that the second statement must execute last; the other two may execute immediately. The language has no method of sequential execution of statements determined by statement ordering.

```
def abc x:integer y:integer returns
integer =
let
 var a, b, c: integer;
 assigns
     a = x + 1;
     b = a * c;
     c = y / 11
in
 b - 2;
```

Figure 2. Example of statement dependencies.

3.2.1.1. Iteration and Recursion

There are a number of issues associated with loops. In a very simple loop - for example initialisation of all components of an array to some constant value (figure 3) - each iteration of the loop is completely independent, and all iterations can be accomplished in parallel.

```
for a from 1 to 100 do
 array[a] = 3.1412
returns
 array;
```

Figure 3. - Parallel array assignment

There are a few alternatives when generating code for a parallel loop. A large number of nodes can be executed in parallel to accomplish the initialisation, or the initialisation can be assigned to a single node, to avoid saturating the processors, and performed serially. It is likely that parallel loops will be unrolled and executed in *clumps*. These clumps will reduce the potential concurrency, but will improve performance by not generating excessive concurrency. The presence of excessive concurrency may increase the computation time by needlessly consuming processing elements and network bandwidth, and is therefore a waste of resources The *clumping* method will still provide significant concurrency but at a reduced cost.

For more complex loops, each loop iteration may have a data dependency from the previous iteration (figure 4). This is handled by the IDA **next** construct:

```
while y < ... do
 ...
 ...
 next x = x + f(y);
 ...
returns
 ...
```

Figure 4. - Successive iteration dependencies

In this example, the variable called x refers to the current loop iteration, unless it is an assignment of the form **next** x =, which refers to the (distinct) variable x in the successive iteration. In this way, data dependencies between successive iterations can occur without breaking the single-assignment rule. In this type of loop, the **next** x = becomes a serializing point: references to x in the successive iteration cannot continue until the **next** ... has been executed. However, this form of code construct is often optimised to permit independant executions of the called function to

execute simultaneously, with a tree of addition operations calculating the result of the `next x =` expression.

3.2.1.2. I-Structures

Id Nouveau is basically a functional language with some additional structures, called I-Structures [17], which are non-functional but still deterministic. The functional part of Id Nouveau consists of a number of definitions and expressions. Definitions bind names to values. Expressions may be arbitrarily complex, and may contain (among other things) function applications, conditionals and loops. I-Structures are provided in an attempt to remove much of the copying which occurs in functional languages, without destroying the determinacy of functional programs. I-Structures allow complex data structures to be created. If an element is accessed before it is defined then the read is deferred until the write has completed. Each element of the structure may be read many times, but may only be written once. Thus I-Structures preserve the dataflow semantics which apply to simple variables in the functional part of Id Nouveau.

3.2.2. Language Extensions

3.2.2.1. Input / Output

Input and output are essential properties of all conventional and commonplace programming languages. This is usually not true for dataflow languages. Originally dataflow languages did not permit any form of input/output. More recent languages, such as SISAL, have allowed the result to be returned to the standard output when the program has completed. Id Nouveau does not provide the ability to read data dynamically. This cannot be considered satisfactory for a usable language. The extension to Id Nouveau provides a set of system routines that allows the programmer to access both the terminal and files. Standard operations include *open, close, read, write,* etc that are available in common high-level programming languages. These routines are not enough to successfully manipulate input/output however. An additional language construct is required - sequential execution. Note that these operations are neither functional nor deterministic.

3.2.2.2. Sequential Execution

Sequential execution allows the programmer to implicitly instruct the compiler to produce code that will be executed serially. Neither Id Nouveau nor SISAL permit this. Both these languages insist that the order of the source lines does not indicate the order of execution. To provide sequential ordering of instructions in either language requires the programmer to explicitly specify dummy dependencies between the operations that they wish to be executed in order. This means that the body of all function invocations are executed in parallel but each function call is guaranteed to wait till the previous (in source line order) has finished before beginning execution. This sequential operation only applies to a single lexical level so that as much concurrency as possible can be attained.

The **sequence** construct permits input/output to behave as the programmer might expect on a sequential architecture (with inherently sequential device hardware). A simple code segment is shown below. However, it is believed that the programmer should restrict the usage of file I/O to the initiation and termination of their programs. This reduces the loss of concurrency. Terminal I/O is less likely to be restricted due to its interactive nature.

```
def writeout = sequence
  = open outfile "filename";
  = writeln outfile a+b "hello world!";
  = close outfile;
  = open infile "filename";
  = readln infile c;
  = writeln stdout "c = " c;
  = close infile
in
  ();
```

Figure 5 - IDA sequential code sample

3.2.2.3. Streams

A powerful feature of SISAL is the data type *stream* [18]. A stream is theoretically an infinite list of heterogeneous data items. SISAL, however, restricts this definition to provide only homogeneous data types for its elements. Streams are not available in Id Nouveau. The programming usage of streams for producer/consumer constructs and file input/output makes this data type (and its associated operators) desirable, so they are included in IDA.

3.2.2.4. Declarations

Id Nouveau originally did not contain strict data declarations, but instead relied on programmer intuition to ensure that type conflicts did not occur. The RMIT version uses a Pascal-like format for declaring types and variables. There were two main reasons for adding fully declared types and variables to Id Nouveau. Most important was the desire of efficient compiled code, and declared variables and types held better prospects for more accurate and simpler code optimisations. Secondly, the usage of variables in a declarative manner was thought to promote faster application development and less debugging due to type conflicts that a typed compiler easily reports. This problem is exacerbated by the lack of adequate debugging environments.

3.2.2.5. The IDA Compiler

The IDA compiler consists of two stages. The first stage compiles IDA source into the intermediate form IF1 (see section 3.3). The second, translates IF1 into the dataflow assembler. By compiling to IF1, IDA is portable, as it can be executed on any of the architectures that have an IF1 translator.

The compilation from IDA to IF1 employs a simple one-pass recursive descent [19] methodology. During this stage, a source level representation of the program, called a *program graph*, is created. Once the parsing has completed successfully, the program graph contains an accurate and complete representation of the source. Code optimisations such as constant folding, sub-expression elimination and code propagation can now be performed on the program graph. Code generation proceeds after all optimisations have been completed.

The compilation process transforms the program graph into IF1 types, function definitions and object code. The advantage of using IF1 has been mentioned earlier. It lies in the ability for several different languages to compile to the same standard form. Such a system permits an computer architect to merely implement an IF1 translator and have software available for immediate use from both the SISAL and (now the) Id Nouveau/IDA parallel community. The disadvantage of compiling to IF1 is conforming to the node boundaries defined by IF1. These are restrictive as the RMIT architecture

supports several of these operations directly, but IF1 forces additional constructs to be planted by the compiler to perform the same function[1] .

3.3. IF1

IF1 [15] is an intermediate graph form developed at Lawrence Livermore National Laboratories. Data in this form is usually the result of a sisal compilation before it is translated into assembly language or binary. The translation from IF1 to the RMIT dataflow assembler is often a 1:1 mapping. IF1 compound nodes and stream operators are more complex. The difficulties with stream operation nodes are caused by the different semantics of IF1's streams and the RMIT architecture's. The RMIT architecture permits heterogeneous stream elements to be accessed in a sequential manner, via *head, tail, cons,* etc nodes. However, IF1 assumes that streams are homogeneous random access data structures. This language feature requires the translator to generate code that traverses the stream to the required element and then performs the necessary operation. Such a procedure is unlikely to be efficient due to the amount of token copying involved in head/tail operations with the result that streams would be operated on by reference as stored objects, for which random access is supported.

The IF1 translator builds a complete internally representation of the program in *dataflow graph* form. Optimisations that are hardware dependent can then be carried out if necessary. This representation is traversed and the assembly code is produced. Code is only produced for accessed routines providing automatic dead code elimination. The more sophisticated IF1 nodes require a different approach. When a compound IF1 node is being generated, the translator must resolve complex implicit dependencies between sub-graphs of each compound node. The more complicated atomic nodes may require an inline to be planted, which is supported by the macro assembler

3.3.1. Predefined System Nodes

The RMIT architecture instruction set includes several complex nodes that are not expressible in IF1. For example, the trigonometric functions are directly supported by the hardware. Achieving access to these hardware instructions directly is a problem that has several possible solutions. These include augmenting IF1 to support a new node for each instruction; having the IF1 translator detect the names of these special nodes as functions when called and substitute the architecture's instruction; or completely ignore the existence of the hardware instructions and require the compilers to plant discrete code to perform existing operations.

The later of these possibilities is clearly unsatisfactory. The first solution is the least portable, but requires modification to a standardised language. The second solution is the most portable, but may cause name resolution difficulties. This solution was selected due to its portability. The name conflict can be resolved by ensuring the user does not declare a function of the same name. If this is done, then the user declared function must be planted.

3.3.2. Stored vs Transmitted Structures

A difficulty with purely functional languages is achieving efficiency. To illustrate this (see figure 6), in the data flow environment, to average the first and last values of an array, the array must be generated in total, and transmitted over the communication network to a node which duplicates the array, sending one copy to a node which

---

[1] An optimising IF1 translator should correct this problem.

extracts the first element, and another copy to a node which extracts the last. The output from these two nodes then go to an adder node, and then a division node, where the sum is divided by the literal 2

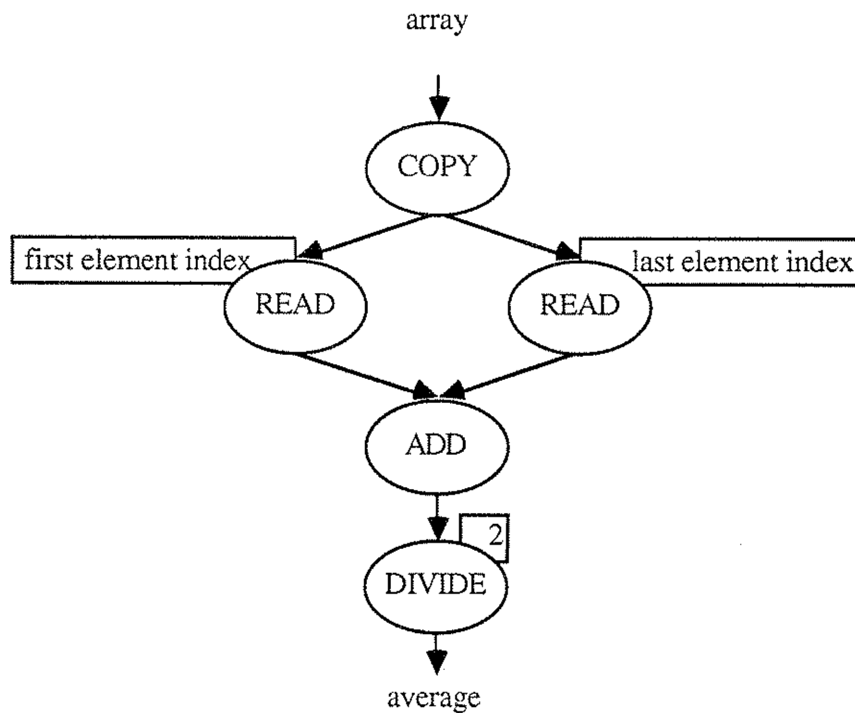


Figure 6. - Average of the first and last elements of an array

This is inefficienct because an excessive degree of data copying is incurred when providing the entire array to both of the array read nodes. With large arrays, this copying is a significant overhead. The advantages of functionality are the lack of state. Maintaining state is a area of concern due to the concentration of operations in a small area. Such concentrations of effort result in considerable restrictions on the amount of concurrency exploitable from the code segment.

In dataflow, the solution to this inefficiency is to create a *structure store*. This is a memory module that can be accessed by special load and store nodes in the dataflow graph. This improves efficiency (as in the above example), but it also introduces problem. The introduction of a structure store, although distributed across the processors, creates potential bottle-necks on frequently referenced data.

As updating, for example, of array components may occur at indeterminate times, the results of execution may themselves be indeterminate due to possible multiple assignment. To prevent this, the structure store can be operated in two modes:

(1) Classical mode, where each memory cell may be read and written multiply, and

(2) Single assignment mode, where each cell is either undefined or has been assigned exactly once. Should a cell be undefined upon an attempted read, the read is suspended until that specific cell has been written, at which time all suspended reads are honoured.

In the IDA language implementation, the first mode is used only by the system, debuggers etc (for example as execution counts in profiling). All user-controlled data is

mapped onto single-assignment cells. Thus, an expression which references an array component value which is undefined will suspend until that component becomes defined. In single-assignment mode, any attempt to assign to an already assigned cell generates a run-time error.

This single assignment property preserves certain aspects of dataflow programs which are desirable. In particular, the results of computation will be *determinate* (that is, repeated execution on a valid environment with matching data will lead to matching results). This approach is not strictly functional - operators in an expression cannot be textually substituted by their results at any time, as their arguments may not be defined. However, it is *functional* given that the computation terminates. This property guarantees a clean programming methodology.

IF1 data structures are likely to use the first mode of the structure store. This is because IF1 is a functional language, and does not attempt to write to a data location multiple times. However, an optimisation is available that uses multiple writes as a means of reducing the amount of structure copying inherent in IF1.

3.3.3. Error Handling and Debugging

Error handling is non-trivial due to eager evaluation. For instance, when evaluating both branches of an if expression, one branch may generate an error (divide by zero ?). At this point it is unknown which branch is legal. Interrupt driven error handling would be inappropriate here as the interrupt may not be required (should the alternate branch be the legal selection). Alternatively, using inline error trapping routines is inefficient. Other schemes, such as error propagation also have difficulties in a functional environment. At this stage, error trapping is favoured as the additional inefficiencies execute in parallel, and should note significantly effect the critical path.

The are also many issues involved with debugging a dataflow graph. Of major significance is the lack of any state information. This arises from the asynchronous nature of the processing elements. To stop all of the processors and examine the state of execution is deemed difficult and worse, potentially indeterminant. Another possibility is the planting of inline debugging code in the dataflow graph. As with error handling, this inefficiency can be executed in parallel and not imped the execution time significantly[1].

3.3.4. In-Situ Updating

The in-situ updates techniques required by the IDA language are not supported by IF1. This is due to IF1's strict data structures (the structure is not available until all component fields are defined). IF2, a superset to IF1 that includes explicit memory management [16], overcomes this difficulty however. At this time, an implementation of IF2 is deemed unwarranted until more resources become available.

Therefore, a new IF1 node is proposed. This node permits non-strict data structures in a manner similar to IF2 by using the RMIT architecture's structure store to contain the data. An implementation of IF2 would be the ideal solution however.

4. CONCLUSION

This paper has discussed issues in the design and implementation of a variant to MIT's Id Nouveau language, IDA, that has been developed at RMIT. Several issues related to

---

[1]assuming that there are sufficient system resources.

this design are significant. Of particular importance is the ability to perform dynamic file operations whilst still maintaining a reasonable degree of concurrency during program execution. Another achievement is the portability of the new language by using a standard intermediate form that is capable of executing on a range of diverse hardwares. The language permits source level comparisons of both the architectures themselves (particularly important for the dataflow machines) and of the compilers and the efficiency of their code generation, for all the architectures.

5. BIBLIOGRAPHY

1. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Dataflow Processor", Proc 2nd Annual Symposium Computer Architecture, New York, May 1975

2. J.B.Dennis, G.A. Broughton, and C.K.C Leung, "Building Blocks for Dataflow Prototypes", Proc 7th Annual Symposium Computer Architecture, La Boil, France, May 1980

3. J.B. Dennis, G.R. Gao, and K.W. Todd, "Modelling the Weather with a Dataflow SuperComputer", IEEE Trans. Computers, Vol -33, No 78, July 1984, pp 592-603

4. Arvind and R.A. Iannucci, "A Critique of Multiprocessing von Neumman Style", Proc 10th Annual Int'l Symposium Computer Architecture, Stockholm, June 1983, pp 426-436

5. Arvind and K.P. Gostelow, "The U-Interpreter", Computer, Vol 15, No. 2, Feb 1982, pp 42-50

6. J. Gurd and I. Watson, "Data Driven Systems for High Speed Parallel Computing - part 2: Hardware Design", Computer Design, July 1980, pp 97-106

7. G.K. Egan, "Dataflow : Its Application to Decentralised Control", Ph. D. Thesis, Department of Computer Science, University of Manchester, 1979

8. D. Abramson and G.K. Egan, "The RMIT Data Flow Computer : A Hybrid Architecture", Royal Melbourne Institute of Technology Technical Report, TR-112-057R, 1987

9. G.K. Egan, "The RMIT Data Flow Computer : Token and Node Definitions", Royal Melbourne Institute of Technology Technical Report, TR-112-60, 1988. INTERNAL RELEASE ONLY

10. D. Abramson and G.K. Egan, "An Overview of the RMIT/CSIRO Parallel Systems Architecture Project", Proc 11th Australian Computer Sciences Conference, Brisbane, 1988

11. R. Nikhil, K. Pingali and Arvind, "Id Nouveau", Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts

12. Kazanori Ueda, "Guarded Horn Clauses", Doctor of Engineering Thesis, University of Tokyo, Graduate School, 1986

13. McGraw, et al, "SISAL : Streams and Iteration in a Single Assignment Language, Language Reference Manual", Lawrence Livermore National Laboratories, M146

14. M. Rawling and C.P. Richardson, "The RMIT Data Flow Computer : DL1 User's Manual", Royal Melbourne Institute of Technology Technical Report, TR-112-058R, 1987

15. S. Skedzielewski and J. Glauert, "IF1 - An Intermediate Form for Applicative Languages", Lawrence Livermore National Laboratories, July 1985

16. M. Welcome, S. Skedzielewski, R.K. Yates and J. Ranelletti, "IF2 - An Applicative Language Intermediate Form with Explicit Memory Management", Lawrence Livermore National Laboratories, December, 1986

17. Arvind, R.S. Nikhil and K.K. Pingali, "I-Structures : Data Structures for Parallel Computing", Computation Structures Group Memo 269, MIT, Laboratory for Computer Science, Feb 1987

18. K.S. Weng, "Stream Oriented Computation in Recursive Data Flow Schemes", Technical Memo 68, Laboratory for Computer Science, MIT, Oct 1975

19. A.V.Aho, R. Sethi and J.D.Ullman, "Compilers - Principles, Techniques and Tools", Addison Wesley, 1986

20. Arvind, D.E. Culler, R.A. Iannucci, V. Kathail, K. Pingali and R.E. Thomas, "The Tagged Token Dataflow Architecture", Laboratory for Computer Science, MIT, July 1983