ALICE

A MULTI-PROCESSOR REDUCTION MACHINE FOR THE PARALLEL EVALUATION OF APPLICATIVE LANGUAGES

John Darlington and Mike Reeve

Department of Computing,
Imperial College of Science and Technology,
180 Queens Gate,
London SW7 2BZ

ABSTRACT

The functional or applicative languages have long been regarded as suitable vehicles for overcoming many of the problems involved in the production and maintenance of correct and reliable software. However, their inherent inefficiences when run on conventional von Neumann style machines have prevented their widespread acceptance. With the declining cost of hardware and the increasing feasibility of multi-processor architectures this position is changing, for, in contrast to conventional programs where it is difficult to detect those parts that may be executed concurrently, applicative programs are ideally suited to parallel evaluation.

In this paper we present a scheme for the parallel evaluation of a wide variety of applicative languages and provide an overview of the architecture of a machine on which it may be implemented.

First we describe the scheme, which may be characterized as performing graph reduction, at the abstract level and discuss mechanisms that allow several modes of parallel evaluation to be achieved efficiently. We also show how a variety of languages are supported.

We then suggest an implementation of the scheme that has the property of being highly modular; larger and faster machines being built by joining together smaller ones. Performance estimates illustrate that a small machine (of the size that we envisage would form the basic building block of large systems) would provide an efficient desk-top personal applicative computer, while the larger versions promise very high levels of performance indeed. The machine is designed to be ultimately constructed from a small number of types of VLSI component.

Finally we compare our approach with the other proposed schemes for the parallel evaluation of applicative languages and discuss planned future developments.

INTRODUCTION

It has become increasingly evident that the conventional approach to programming using high level languages such as ALGOL or Pascal is inadequate to cope with the size or complexity of the applications now being tackled. Critics of these languages trace their shortcomings to their connection with the von Neumann model of computation and highlight their non-mathematical behaviour, in particular the lack of referential transparency.

In contrast, the applicative or functional languages (e.g. HOPE [BMS80] or KRC [Tur80]) are based on firmer mathematical foundations. This, it is claimed, makes programming and program understanding a much simpler activity; a point that is made forcibly by Backus in his ACM Turing Award Lecture [Bac78].

Despite the numerous advantages of applicative languages their poor performance on conventional machines has meant that they have not gained widespread acceptance. This poor performance may be attributed to their natural mode of evaluation being far removed from the von Neumann model. In an attempt to alleviate this problem alternative computer architectures are being sought. The applicative languages are natural candidates for parallel evaluation since by definition they are free from side effects and thereby permit the independent evaluation of sub-expressions.

Recent advances in semiconductor technology have dramatically increased the integration densities that can be achieved for digital circuits. At the same time the manufacturing costs have fallen equally dramatically. The rate and scale of these changes have been such that a "VLSI crisis" has now arisen, characterized by the questions 'How do we design chips with such high gate densities ?' and, more importantly from our point of view, 'What devices do we fabricate ?'.

The diminished costs of mass production seem to demand that future systems be constructed from many instances of some standard building block (or at least have their components selected from some very small set). One of the most obvious candidates for such a universal element is the single chip microcomputer (c.f. Baron's Transputer [Bar78]). Whatever the building block it suggests that future systems should be composed of large numbers of distributed concurrent activities; this too is a concept far removed from the von Neumann model of computation.

In this paper we provide an overview of a scheme for the parallel evaluation of a wide variety of applicative languages and explore its architectural implications. The scheme, which we call ALICE (Applicative Language Idealized Computing Engine), promises to provide an efficient method of evaluating applicative programs that may be implemented with a small number of types of VLSI component. We therefore regard our work as part of the inevitable fusion of the research in the programming language, computer architecture and VLSI fields.

ABSTRACT SCHEME

Parallel Reduction Of Applicative Languages

As we shall see our scheme is designed to accomplish the parallel reduction of a wide variety of applicative languages. We will introduce it using an example written in a first order recursion language loosely based on NPL [Bur77]. Programs in such a language consist of sets of equations defining functions.

The following program computes Factorial(n) (n > 0).

```
Factorial : Integer  ->  Integer
   Factorial(n)   <=  FactB(1, n)

FactB : Integer x Integer  ->  Integer
   FactB(i, i)    <=  i
   FactB(i, i+1)  <=  i+1
   FactB(i, j)    <=    FactB(i, mid)
                      * FactB(mid, j)
   where  mid  ==  IntegerDivide(i+j, 2)
```

Consider the expression Factorial(5). Its value may be computed by applying reductions to the associated graph.

A conventional evaluator reduces one sub-part of the expression graph at a time; either the leftmost-innermost (call by value) or the leftmost-outermost (call by name).

The constraint that we reduce one expression at a time is completely artificial and is imposed purely because of the sequential nature of the von Neumann machines on which the current evaluators are implemented. In fact because of the independence of sub-expressions in applicative languages, there is no reason why all the available sub-parts of a graph should not be reduced in parallel. Applied to Factorial(5) this full-substitution approach [Man74] would yield the computation sequence illustrated in Figure 1.

A computer system capable of physically realizing such parallel evaluation would offer a significant increase in performance over today's sequential evaluators.

Modelling Graph Reduction

The graph of an expression is represented by a collection of packets. Each packet represents one node of the graph and the arcs extending downward from that node, together with some control information required by the evaluation mechanism. As shown in Figure 2 a packet consists of three primary fields and three secondary fields. The primary fields contain the information required to represent a node, while the secondary fields contain the control information required by the evaluation mechanism.

The functions of the three primary fields are as follows:

i) Identifier
This field contains an identifier unique to the packet (with respect to all other packets in existence during the life of the packet) and provides a name by which the packet may be referenced.

ii) Function
This field contains the function associated with the node represented by the packet.

iii) ArgumentList
This field contains the identifiers of the packets representing the offspring of the node (i.e. the arguments of the function). When the packet is employed to represent a numeric value the ArgumentList field is replaced by the Value field that contains the binary representation of the value.

The functions of the secondary fields will be introduced as required. However, it should be noted that the Status field contains several sub-parts.

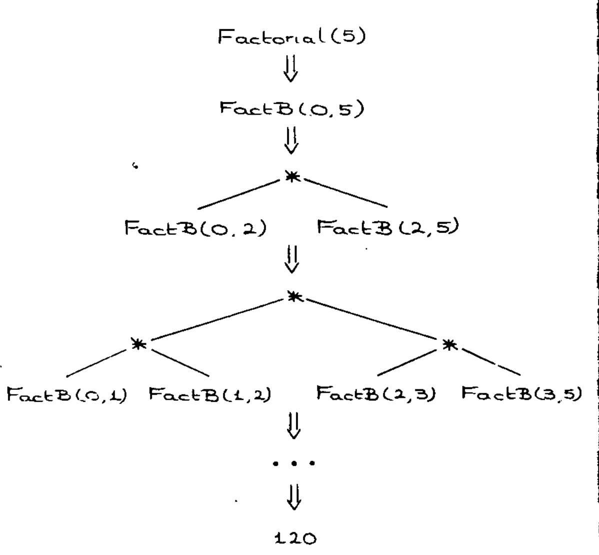

Figure 1: Parallel Evaluation

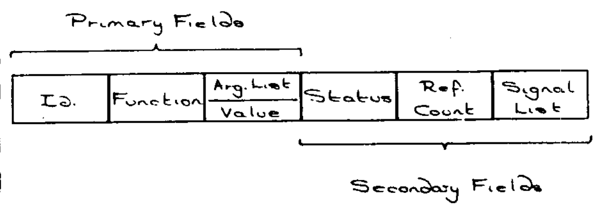

Figure 2: A Packet

In the parallel evaluation of Factorial(5) each graph was transformed into the next by applying one reduction to every reducible node present. The transition from one collection of packets to the next is accomplished by replacing the packets representing reducible nodes by those representing the, suitably instantiated, right hand sides of the appropriate rewrite rules.

Figure 3 shows the packet scheme equivalents of the graphs and reduction steps of Figure 1 (in order to reduce the number of packets that need be illustrated, the shorthand notation [N] will sometimes be used to denote the identifier of the packet that represents the integer value N).

Note that when a packet reconfigures the topmost packet of the resulting collection adopts the identifier of the rewritten packet, since the sub-expression the packet represents is referenced by that identifier.

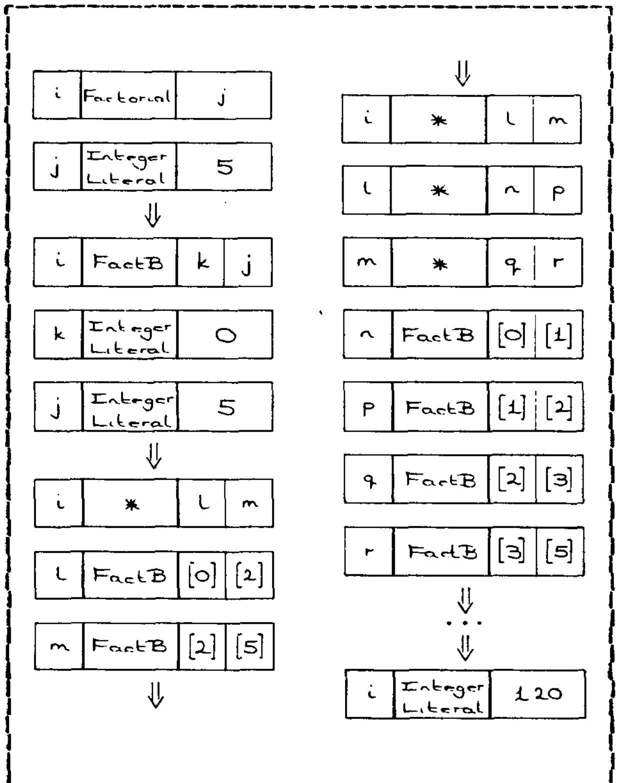

Figure 3: The Packet Scheme

Waiting For Arguments

For a node to be reducible it must satisfy two requirements. Firstly it must be associated with a reducible function (i.e. one for which there are rewrite rules) rather than a constructor function. Secondly the arguments must be of a form that matches the left hand side of some equation in the function definition.

The following program for appending an item onto the end of a list illustrates the implications of this second requirement.

```
Append(x, nil)    <=   x::nil

Append(x, e::L)   <=   e::Append(x, L)
```

When a packet whose Function field contains Append is reduced the appropriate rewrite rule is determined by which of the two (List) constructor functions is to be found as its second argument. Thus before any packet whose Function field contains Append may be reduced the Function field of the packet representing the root node of the sub-graph associated with its second argument must contain a (List) constructor function. As the first argument does not appear as a constructor function on the left hand side of any rewrite rule, its form does not constrain the reduction of a packet whose Function field contains Append.

The situation where the Function field of some argument packet does not contain a constructor function when it is required to do so may be handled in one of two ways.

i) The packet could repeatedly poll its failing argument packets until the Function field of each contains a constructor function.

ii) The failing argument packets could be made responsible for informing the packet when they become of the correct form.

Approach (ii) is adopted since (i) is wasteful of processing resources.

The **SignalList** field and the **PendingArguments** sub-part of the **Status** field are used to implement this mechanism.

If the Function field of an argument packet does not contain a constructor function when it is required to do so, the interested packet leaves its identifier in the SignalList field of that packet. Having checked the form of any other arguments that are required to be constructor functions, the packet sets the PendingArguments sub-part of its Status field to indicate that it is asleep waiting for the appropriate number of arguments to coalesce to constructor functions.

When the Function field of a packet is rewritten to a constructor function, wake-up signals are sent to any packets whose identifiers appear in the SignalList field.

Once a packet has received signals from all of its failing argument packets it then represents a reducible node.

Figure 4 shows the operation of this mechanism during the evaluation of Factorial(5).

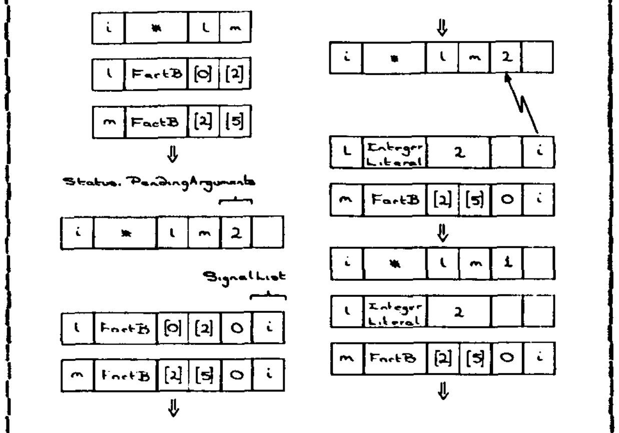

Figure 4: Waiting For Arguments

Lazy Evaluation

Consider the following program for building the infinite list of integers.

```
Numbers :  ->  List Integer
   Numbers <= 1::IncrementByOne(Numbers)
```

```
IncrementByOne :     List Integer
                  -> List Integer

    IncrementByOne(n::L)

        <=  (n+1)::IncrementByOne(L)
```

Thus far we have only described an eager evaluator. As such it would generate elements of the list without reference to their need. However, in general:

- only some finite initial segment of such a list is required,
- the elements are consumed sequentially by a stream function [KaM77].

Consequently the resources may become saturated with the computation of either unwanted elements or elements that will not be consumed for some time.

By making the evaluation of Numbers lazy (i.e. such that a node at which it appears does not reduce until explicitly requested to coalesce to a constructor function; see [FrW76], [Hen76], [Hen80]) the possibility of such resource saturation is eliminated.

Lazy evaluation is implemented as follows. When packets representing an expression that is to be the subject of lazy evaluation are generated those whose Function fields contain a reducible function are marked as being lazy and not-yet-required (via two sub-parts of the **Status** field) and as such are not considered to be candidates for reduction. When such a packet is inspected to determine whether or not the node it represents is associated with a constructor function, the signal request mechanism described above changes its status from not-yet-required to required thereby making it a candidate for reduction. The lazy tag in its Status field is used to propagate the laziness to the packets generated as a result of its reduction.

A purely lazy evaluator, although safe, may be less than optimally efficient since unutilized resources could be used to pre-evaluate expressions where appropriate. Thus our scheme provides for both eager and lazy evaluation and allows the user to determine which of the two modes is appropriate to each expression evaluated. The user could either be the programmer (via source code annotations) or some form of automated anotator (c.f. the work of Mycroft [Myc80] and Arvind to detect when eager evaluation is safe).

Constrained Eager Evaluation

Consider the following expression, where COND is the traditional **if ... then ... else ...** function:

```
COND(P, Q, R)
```

In an eager evaluation each of the sub-expressions P, Q and R would be evaluated in parallel. However:

- only one of Q and R is required,
- only one of Q and R may be defined (e.g. Q might be the general case processor, R the exception case processor and P the exception case detector).

Demanding that the alternatives be subject to lazy evaluation removes the problem of spurious evaluation. However, if after selection it were permissible for the appropriate alternative to be the subject of eager evaluation, then some potential parallelism would be lost.

Thus the evaluation of Q and R must be delayed until P has selected the appropriate alternative irrespective of Q and R's mode of evaluation. The above is implemented as follows. When the collection of packets associated with COND(P, Q, R) is generated, those representing Q and R are marked as being suspended (via a sub-part of the **Status** field). COND is made to send activate signals to the packets representing the selected alternative. Thus:

```
COND(TRUE,  X, Y)  =>  activate X

COND(FALSE, X, Y)  =>  activate Y
```

Languages Supported

The variable free languages (e.g. Turner's Combinators [Tur79] or Backus's FP Languages [Bac78]) can be accomodated very simply. Here there is a fixed set of rewrite rules (i.e. those that define the Combinators or the FP Operators) that are applied to the collection of packets that represents the program source.

The higher order functional languages can be compiled into variable free languages (c.f. the work of Turner [Tur81]), however, we can also support them directly.

The higher order functional languages are distinguished from the first order ones by the ability to pass functions as arguments and return them as values. Two events are associated with such functions, **definition**, where function valued objects are formed and **application**, where such objects are applied to appropriate arguments.

Hence,

$$F(X, Y) \Leftarrow \lambda z \,.\, (X * Y) + z$$

defines a function, F, that returns a function as its value. Thus F(2, 3) is the function that adds 6 to any number and F(2, 3) (4) is 10.

Function definition is modelled by the **LAMBDA** operator which has two arguments, the first being a list of the bound variables and the second the packets representing the function body. Bound variables are referenced in the function body by their position in the bound variable list. The bound variable list is initially represented by a single place marker packet. Free variables in the function body refer to the packets representing their values at function definition time.

Thus evaluating F(2, 3) would produce the collection of packets shown in Figure 5.

The packets representing the function body may be divided into two categories.

i) Those packets that represent the sub-parts of the function body that do not contain references to any bound variable and, therefore, may be evaluated at function definition time (e.g. packet 'n' in Figure 5).

ii) Those packets that represent the sub-parts of the function body that reference bound variables. These form a prototype to be used at function application time (e.g. packets 'm' and 'p' in Figure 5).

Application is modelled by the function **APPLY(f, args)** which takes a function, **f**, and applies it to a set of arguments, **args** (supplied as a list).

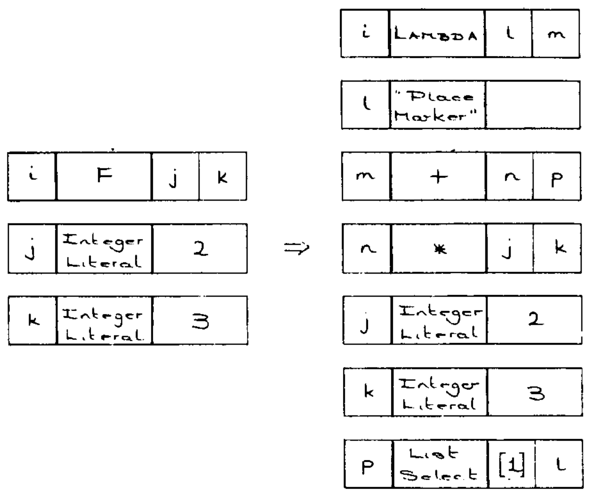

Figure 5: Evaluation Of F(2, 3)

An **APPLY** packet reconfigures by copying the sub-parts of the function body that contain references to bound variables. During this copying any references to the bound variable list place marker are replaced by a reference to the actual argument list (c.f. parameter passing). When this has been done the instantiated copy of the function body can be executed in the normal way.

Figure 6 shows the application of F(2, 3) to 4.

"**Don't care**" **nondeterminism** (c.f. Dijkstra's Guarded Commands [Dij75]) is provided by allowing more than one rewrite rule to apply for some value(s) of a function's argument(s). A variety of mechanisms are provided for choosing which rule to apply in these circumstances (e.g. random selection, most general / most specific case, availability of arguments, etc.). This feature is directed towards Artificial Intelligence applications involving Production Systems [DaK77] and current work on functional operating systems [Hen81]. For example, consider the following function.

```
Merge(x1::L1, L2)   <=   x1::Merge(L1, L2)

Merge(L1, x2::L2)   <=   x2::Merge(L1, L2)
```

The intent of the above is as follows. If the first argument has coalesced to a constructor function and the second has not, then the first rewrite rule should be applied. Conversely if the second argument has coalesced to a constructor function and the first has not, then the second rewrite rule should be applied. If both arguments have coalesced to constructor functions, an arbitrary selection should be made between the two rewrite rules.

The logic based programming languages (e.g. [Kow74], [CMG81]) are supported by a facility that permits a packet to be treated as a variable by allowing a reduction to have the side-effect of assigning new contents to any argument packet. This supports the **logical variable** that may be instantiated through unification [Kow79].

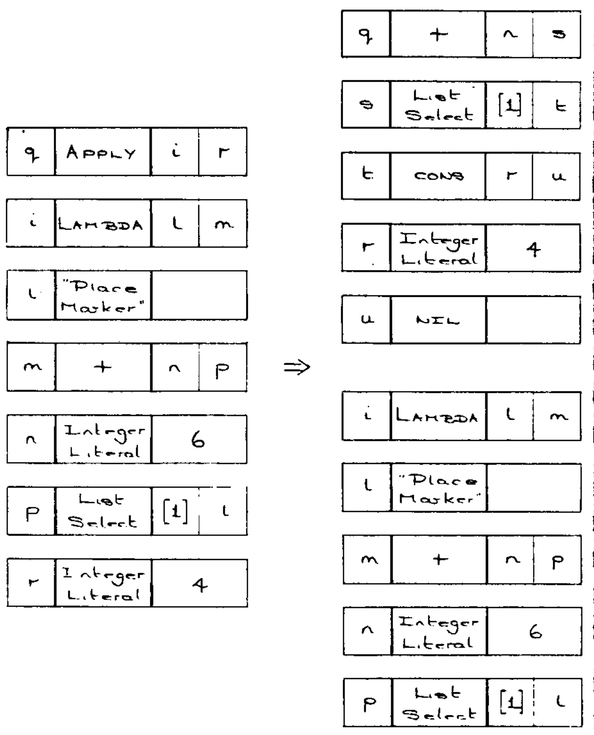

Figure 6: Application Of F(2,3) To 4

For example, given the relation:

```
Times(u, v, w)
```

that holds when u * v = w, assume that it is to be used to generate w given u and v. Figure 7 illustrates a typical reduction step involving a packet whose Function field contains Times.

Combining this with "don't-care" nondeterminism enables our scheme to implement directly the language described in [ClG81].

A variety of imperative languages are also supported. The ability to treat packets as variables allows the single assignment applicative languages associated with the data flow schemes to be executed directly. Furthermore, combining the ability to treat packets as variables with the facility to serialize the computation permits the execution of the conventional (von Neumann) languages.

A compiler target language, [Ree81], has been designed to provide the facilities for language implementation described above. This language defines the abstract machine that ALICE will implement. At present we are developing a compiler from HOPE to this language.

Garbage Collection

Our scheme is such that no packet may be self referential (i.e. expression graphs may not contain cycles). Thus a reference count is sufficient to recognize a packet that has become dereferenced. The **ReferenceCount** field maintained in each packet is used to implement this style of garbage collection.

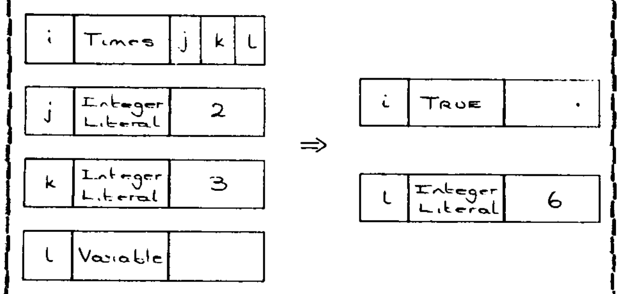

Figure 7: Resolution

Unfortunately, requiring that expression graphs do not contain cycles leads to the inefficient evaluation of data structures such as X <= 1::X; as can be seen from Figure 8. Figure 8a shows the inefficient evaluation forced upon our scheme while Figure 8b shows the more efficient evaluation involving a cyclic graph. At present we are investigating a more complex garbage collection scheme that will permit cyclic graphs.

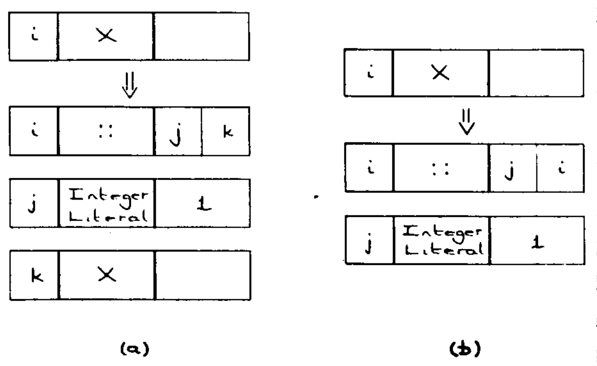

Figure 8: A "Cyclic" Data Structure

The great advantage of being able to perform garbage collection via a reference count is that it can be done concurrently with program evaluation and in a completely distributed fashion.

ABSTRACT ARCHITECTURE

Components

At the abstract level the machine required to implement the scheme described above can be envisaged as a pool of packets and a collection of packet processing agents (Figure 9).

An agent executes the following sequence of activities to model the reduction of a node:

1. Remove some processable packet from the packet pool.

   i.e. a packet whose Function field contains a reducible function and whose Status field does not indicate that it should not be processed (e.g. because it is asleep or part of a suspended expression, etc.).

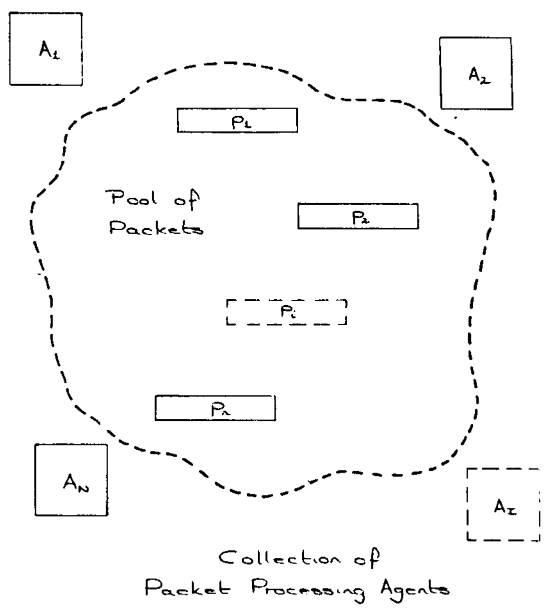

Figure 9: The Abstract Architecture

2. Decide whether it is reducible.

   i.e. check whether the packets associated with any arguments required to be constructor functions are of the correct form.

   If any argument packets are not of the required form,

   a) Leave the identifier of this packet in the Signal field of each failing argument packet.

   b) Mark this packet as being asleep pending the appropriate number of wake-up signals.

   c) Restore this packet to the packet pool.

   d) goto 1

3. Determine the appropriate rewrite rule.

   i.e. match this packet and its argument packets (if any) with the left hand side of some rewrite rule.

4. Generate the packets representing the right hand side of the rule and deposit them in the packet pool.

   i.e. for each,

   a) Acquire an unused identifier.

      Remember that the packet representing the outermost function on the right hand side adopts the identifier of this packet.

   b) Form the contents of the packet body.

   c) Deposit the packet in the packet pool.

5. goto 1

Although the above implies that all agents are identical and that each has a copy of the rewrite rules associated with all functions, there is no reason why agents could not be allowed to specialize in the execution of particular functions. The types of function that are obvious candidates for implementation via special purpose agents are those that perform input / output and those whose algorithms are directly implementable in hardware.

A large number of the other parallel machine proposals involve a single copy of each function definition being held in some central store (e.g. the instruction store of the data flow machines). In our scheme each agent has a local copy of the function definitions. Although we have thus removed a bottleneck that potentially limits the system's performance, we have introduced the overhead of distributing the associated function definitions before a program can be evaluated. However, this could be avoided by performing this distribution concurrently with program evaluation. Initially the function definitions would be loaded into some central store and the agents would know of no function (i.e. their local function definition stores would be empty). When an agent began to process a packet whose Function field contained a function that it did not know, it would copy the definition from the central store to its local store. Thereafter whenever the agent processed a packet whose Function field contained that function it would have a local copy of its definition (c.f. demand paging).

A simulator (written in Pascal) has been developed for the abstract machine described above. This permits,

- execution of programs written in the compiler target language according to the scheme,
- predictions of performance (by extrapolating the statistical information produced onto the proposed implementations).

IMPLEMENTATION CONSIDERATIONS

Implementing The Abstract Architecture

Two distinct types of access are made to the packet pool, the first is an access directed at a packet with a specific identifier (e.g. to inspect an argument, to send a wake-up signal, etc.), the second is an access directed at an arbitrary member of a particular class of packet (e.g. when a agent requires work it must find a packet whose Function field contains a reducible function and whose Status field does not indicate that it should not be processed for some reason or other). Our proposed machine attempts to separate the mechanisms by which each of these types of access are implemented.

In the physical realization of the abstract architecture a packet's identifier is equated to a global address in a random access store of packets; thus the Identifier field is not present in the physical realization of a packet since the information it would contain is implicit in the packet's location.

A mechanism, that may be compared to the task scheduler and storage management system of a conventional machine, provides each agent with immediate access to the set of processable packets and the set of unused packet identifiers (i.e. the free locations in the packet store). This mechanism is implemented independently of the identifier directed access to the packet store. A constantly circulating slotted communications ring onto which each agent has a one slot window may be regarded as a set with distributed concurrent access. We propose two such rings, one to represent the set of processable packets and the other to represent the set of free packet store locations. Thus, for example, were an agent required to generate a new packet it would simply employ the packet store location whose address was currently in its window onto the ring of free packet store locations.

Although the locations within the packet store are accessed via a global address, it is implemented in a distributed fashion by interconnecting a collection of small segments. The envisaged implementation employs a building block consisting of several processing agents connected to a local memory via a shared bus. The local memory represents one segment of the packet store. Building blocks are assembled into larger systems using an interconnection network. The interconnection network serves both to map the local memories onto the global address space of the system and to share work (processable packets) and free storage (unused packet identifiers) amongst the building blocks. By varying the number of agents connected to each local memory and the size of each local memory, the system ranges from a tightly coupled to a loosely coupled one.

The indentifier directed packet pool / agent traffic is a limiting factor in respect to system performance. Thus the number of concurrent accesses that may be made to each packet and the time per access must be maximized and minimized respectively. The scheme is such that any number of reads may occur concurrently but writes must have exclusive access to a packet. Lockout only occurs at this low level, there is no higher level of packet access control; for example, even when a packet is being processed the contents of its associated packet store location may be accessed (e.g. to be inspected by by those packets that require it to be a constructor function). Thus any number of agents may execute the following program concurrently on the same list.

```
MinOfList : List Integer -> Integer
    MinOfList(x::nil)    <=  x
    MinOfList(x::y::L)
        <=  Min(x, MinOfList(y::L))
Min : Integer x Integer -> Integer
    Min(u, v)   <=  u  if  u < v
                    v  otherwise
```

Factors That Influence Performance

The performance of a system with an abstract architecture of the type outlined above is not related to the throughput of an agent when the number of agents is equal to that which may share access to the packet pool without contention.[1] Under these conditions the performance (in terms of packets processed per second) is given by:

$$\frac{\text{ConcurrentAccessesPermittedToPacketPool}}{(\text{AccessesPerPacketProcessed} * \text{AccessTime})}$$

Consequently none of the proposed implementations makes any attempt to exploit the opportunities for parallelism

---

[1] The number of processing agents that may share access to the packet pool without contention is given by:

$$\frac{\text{Time To Process A Packet}}{(\text{AccessesPerPacketProcessed} * \text{AccessTime})}$$

within an agent (e.g. when determining which rewrite rule is applicable). We contend that the increased number of agents required to ensure the maximum utilization of the packet pool bandwidth is offset by the reduced complexity of each.

A Desk-Top Machine

If the number of concurrent accesses to the packet pool is limited to one, the agents may access the pool via a shared bus. This is to be the case in our system building block. The performance (in terms of packets processed per second) will be given by:

$$\frac{1}{(AccessesPerPacketProcessed * BusTime)}$$

It is envisaged that the agents of our system building block will require approximiately 128 uS to process a packet and that each packet pool access will require less than 1 uS. Although the average number of packet pool accesses per packet processed will depend on the expression being evaluated, analysis of the results of executing a variety of programs on our current ALICE simulator indicates that it is around 6. Thus approximately 20 processing agents will fully utilize the available packet store bandwidth. These estimates indicate that this desk top sized system will be capable of executing approximately 150 thousand rewrites (packets processed) per second; this represents a two orders of magnitude increase in performance over the existing HOPE implementation on a DEC-10 with a KI processor.

A Large Scale Machine

We view the small scale machine described above as a system building block. A range of larger machines may be configured by linking together a number of such units.

A large scale machine is built by joining together a number of small scale machines via an interconnection network that links all of the individual packet pool segments so that they appear as a single global entity and distributes work (processable packets) and storage (unused packet identifiers) across the system. Ideally the load sharing mechanism should be intelligent enough to recognize when it is not worth transfering work or free storage from one node to the next because the overhead of accessing a remote segment of the packet pool will outweigh the benefits of the increased parallelism.

Interconnection networks of the type proposed above are notoriously difficult to implement. Ideally the network should be such that:

- the number of circuits required grows slowly with respect to the number of nodes,
- the potential loss of machine performance due to the drop in network throughput which results from increasing the number of nodes is more than offset by the extra processors added to the system.

In a closely coupled implementation the ideal mechanism, in terms of throughput, is the crossbar switch, however, such devices are not practical for large numbers of nodes because the number of circuits required is proportional to the square of the number of nodes. The recently developed Delta Network [Pat79] seems to provide a reasonable trade off between performance and the number of circuits required.

The type of interconnection network traditionally employed in closely coupled systems considers all nodes to be equi-distant. Thus the transit time is constant and independent of the source and destination nodes. In our application, however, we expect the intelligent load sharing mechanism introduced above to maintain a fair amount of locality of reference. Consequently we are investigating a class of networks for which the transit time is a function of the closeness of the nodes. Examples of networks suited to this more losely coupled view are Hewitt's Apiary [Hew80] and Burton and Sleep's Warren [BuS80].

We estimate that a system consisting of approximiately 4096 nodes (building blocks) could execute in excess of 150 million rewrites (packets processed) per second.

Hardware

The modular nature of the above proposals make them ideally suited to VLSI implementation and initial design studies indicate that the system building block may be constructed from a very small number of types of VLSI component.

A COMPARISON WITH RELATED RESEARCH

General Comparsions

The ability to increase a system's performance by increasing the number of processing agents and the high reliability imparted by requiring that only one agent be operational for a system to function correctly are properties shared with most of the other multi-processor architectures that have been proposed for the parallel evaluation of applicative languages. Furthermore, since packets from any number of sources (jobs) may be present in the packet pool without interfering with each other, any surplus of agents may be counteracted by multi-programming the system. As with the majority of the other machine proposals, run-time type checking can be performed with only a minimal overhead.

Our scheme, we consider, provides a mechanism that supports a wide variety of applicative languages and evaluation strategies. As such it forms a component of our long term goal, a complete applicative programming environment. Another component, a meta-language driven transformation system, is being developed and is described in [Dar81]. This allows programs to be developed systematically from specifications and is completely applicative, both the meta-language and the object language being HOPE. The program transformation tactics it uses, [BuD77], originally designed to improve the efficiency of applicative programs when run on a sequential machine, are equally applicable to improving programs to be run on parallel machines such as ALICE.

Data Flow

Our scheme evolved from a study of the data flow projects particularly that reported in Gurd, Watson and Glauert [GWG80], Arvind, Gostelow and Plouffe [AGP78] and Dennis, Leung and Misunas [DLM79]. However, our scheme supports a much wider range of evaluation mechanisms.

Although data flow is the natural mechanism by which to implement the single assignment applicative languages and thereby perform the high speed evaluation of scalar expressions, the graph reduction model of computation seems to directly support a wider range of applicative languages (particularly the zero assignment languages such as HOPE and Backus's FP).

Furthermore, because in a zero assignment functional language data structures are simply terms involving constructor functions, our scheme accommodates them directly. In contrast, the existing data flow machines have had to introduce additional mechanisms to provide for data structures (e.g token labelling to support arrays [GWG80] or structure memories [Ack77]).

String Reduction

In contrast to those machines which perform string reduction (e.g. [Ber75] and [Mag79]) where all expressions are copied, our scheme performs graph reduction and therefore common sub-expressions can be shared (hence removing the inefficiency of repeatedly evaluating the same expression).

In addition, the proposed machines store the string being manipulated in the elements of a linear array of cells, thus as the string expands and contracts during reduction the contents of the cells must be shifted accordingly. However, the storage management system of our machine allocates and reclaims store (unused packet identifiers) as if it (they) were on a heap, thus storage management operations on one cell do not have the global repercussions that they do on the string reduction machines.

Sleep and Burton [SlB81]

This project also proposes a network of processors to perform graph reduction. However, their system is purely demand driven and, at present, only seeks to exploit the parallelism inherent in divide-and-conquer algorithms.

Also their interconnection network is used soley to distribute work; it maps the logical computation graph onto the physical array of processing agents. The mapping scheme is such that one agent may become associated with more than one node of the computation graph, thus their agents must be multi-tasked. Futhermore, the workload distribution might be such that some agent may have to sit idle until the data it requires has been produced by some other agent. In our scheme the agents conceptually move around the logical computation graph processing one node at a time, thus they are only uni-tasked and therefore no agent may be idle when there is work (reducible nodes; i.e. processable packets) available.

When tasks are distributed across the Sleep and Burton network any data required by each is distributed with it, thus removing the need for remote data accesses during the execution of a task. In our scheme processes (i.e. processable packets) only contain references to data. Thus, although our scheme requires remote data accesses, the total communications overhead is likely to be somewhat less when a large data structure is processed, since in general, each process will only require access to some sub-part of the data structure.

Keller and Lindstrom [KLP78]

Although our scheme was developed independently of that of the Utah group there are a number of similarities between our two projects. At the logical level the similarities in the evaluation mechanisms and the treatment of higher order functions seem to be a result of both schemes having their roots in the graph reduction model of computation. At the physical level the similarities arise from the need to solve the recurring problem of implementing a large random access memory in a distributed fashion.

The Utah scheme exploits parallelism at the level of calls to user defined functions. Our scheme not only exploits parallelism at this level but, employing the same mechanism, also exploits any parallelism to be found in the function body.

As in Sleep and Burton's proposal, the logical computation graph is mapped onto a physical array of processors in such a way that one processor may become associated with more than one node, thus their processing agents must be multi-tasked and thereby more complex than those of our scheme. The work load distribution / idle processing agent problem highlighted in the Sleep and Burton project is also associated with this scheme.

Finally, we feel that our compiler target language facilitates the implementation of a wider range of languages than that of the Utah project.

Tree Machines

Several projects (e.g. The Tree Machine [Bro80]) seek to map the logical computation graph onto a physical tree of processors. However, the majority of these proposals have not yet fully addressed such problems as:

- How the machine copes with computation graphs that contain more nodes than the processor tree (or unbalanced graphs where some branches outgrow the tree while others leave processors unoccupied; thus although the graph may contain fewer nodes than processors it can still not be mapped onto the machine).
- The non-utilization of the processors associated with processes that are waiting for arguments.

In our machine the computation graph is mapped onto the packet pool and conceptually our processors move from node to node via the interconnection network. Thus the topology of the machine puts no constraints on the nature of the computation graph.

Carnegie-Mellon

The concept of a distributed but globally addressed store is shared with the multi-processor architectures developed at Carnegie-Mellon (e.g. Cm* [SFS77] and C.mmp [FuH78]). However, these machines were designed to support interacting sequential processes; thus the user must explicity program concurrency and synchronize access to shared data. Whereas our machine was designed top-down from the applicative programming methodology and exploits the inherent parallelism available during the evaluation of applicative programs.

FUTURE RESEARCH

The current ALICE simulator operates at a fairly abstract level, therefore a second simulator (written in PATH Pascal [CGM78]) is being developed for the building block hardware so as to facilitate more detailed architectural experiments.

As was indicated above, ALICE seems ideal for VLSI implementation and we are actively persuing this avenue of development.

A variety of projects to provide software for ALICE (e.g. compilers, an operating system, a program development system, etc.) are also in progress.

ACKNOWLEDGEMENTS

The scheme we have explained obviously owes a great debt to the data flow work, particularly that reported in Gurd, Watson and Glauert [GWG80], Arvind, Gostelow and Plouffe [AGP78] and Dennis, Leung and Misunas [DLM79]. We have also benefited from the work of Treleaven [Tre80a] and discussions with those who attended the Joint SRC / University of Newcastle upon Tyne Workshop on VLSI : Machine Architecture and Very High Level Languages [Tre80b] and our colleagues in the Department of Computing at Imperial College.

The comments of the referees and Arvind and Jack Dennis have greatly enhanced both the content and presentation of the paper.

Both authors gratefully acknowledge the support of the Science and Engineering Research Council of Great Britain.

REFERENCES

Ack77 Ackerman W. B. A Structure Memory for Data Flow Computers. Laboratory for Computer Science, MIT, LCS/TR-186, 1977.

AGP78 Arvind, Gostelow K. P. and Plouffe W. An Asynchronous Programming Language and Computing Machine. University of California (Irvine) Report, Dept. of Computer Science, 1978.

Bac78 Backus J. Can Programming be Liberated from the von Neumann Style ?. ACM Turing Award Lecture. CACM Vol. 21 No. 8, Aug. 1978.

Bar78 Baron I. The Transputer. In 'The Microprocessor and its Applications', ed. Aspinall D., Cambridge University Press, 1978.

Ber75 Berkling K. Reduction Languages for Reduction Machines. Proc. Second Int. Symp. on Computer Architecture, 1975.

Bro80 Browning S. A. The Tree Machine. Ph.D. Thesis, Computer Science Dept., California Institute of Technology, 1980.

BMS80 Burstall R. M., MacQueen D. B. and Sannella D. T. HOPE: An Experimental Applicative Language. Internal Report, Dept. of Computer Science, University of Edinburgh, 1980.

BuD77 Burstall R. M. and Darlington J. A Transformation System for Developing Recursive Programs. JACM Vol. 24 No. 1, Jan. 1977.

Bur77 Burstall R. M. Design Considerations for a Functional Programming Language. Proc. of Infotech State of the Art Conference, Copenhagen, 1977.

BuS80 Burton F. W. and Sleep M. R. Large Symmetrical Processor Networks with Short Communication Paths. School of Computing Studies and Accountancy, University of East Anglia, Report CS/80/019/I, 1980.

CGM78 Campbell R. H., Greenberg I. B. and Miller T. J. Path Pascal User Manual. Dept. of Computer Science, University of Illinois at Champaign-Urbana, 1978.

ClG81 Clark K. L. and Gregory S. A Relational Language for Parallel Programming. Dept. of Computing, Imperial College, London. This conference.

CMG81 Clark K. L., McCabe F. G. and Gregory S. IC-Prolog Reference Manual. Research Report, Dept. of Computing, Imperial College, London. (Under revision.)

Dar81 Darlington J. The Structured Description of Algorithm Derivations. Invited paper. International Symposium on Algorithms, Amsterdam, 1981.

DLM79 Dennis J. B., Leung C. K. C. and Misunas D. P. A Highly Parallel Processor Using a Data Flow Machine Language. Laboratory for Computer Science, MIT, CSG Memo 134-1, June 1979.

FrW76 Friedman D. P. and Wise D. S. CONS Should Not Evaluate Its Arguments. In 'Automata, Languages and Programming', eds. Michaelson S. and Milner R., Edinburgh University Press, 1976.

FuH78 Fuller S. H. and Harbison S. P. The C.mmp Multiprocessor. Dept. of Computer Science, Carnegie-Mellon University, Technical Report CMU-CS-78-146, 1978.

GWG80 Gurd J. R., Watson I. and Glauert J. R. W. A Multilayered Data Flow Computer Architecture (3rd Issue). Internal Report, Dept. of Computer Science, University of Manchester, 1980.

HeM76 Henderson P. and Morris J. H. A Lazy Evaluator. Proc. Third Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, 1976.

Hen80 Henderson P. 'Functional Programming'. Prentice-Hall, 1980.

Hen81 Henderson P. Working Material for the Lectures of P. Henderson. Advanced Course on Functional Programming and Its Applications, University of Newcastle Upon Tyne, July 1981.

Hew80 Hewitt C. The Apiary Network Architecture for Knowledgeable Systems. Proc. of the 1980 LISP Conference, 1980.

KaM77 Kahn G. and McQueen D. Coroutines and Networks of Parallel Processes. Proc. IFIP Congress 77, North Holland Publishing Co., 1977.

KLP78 Keller R. M., Lindstrom G. and Patil S. An Architecture for a Loosely-Coupled Parallel Processor. Dept. of Computer Science, University of Utah, Tech. Report UUCS-78-105, 1978.

Kow74 Kowalski R. A. Logic as a Programming Language. Proc. IFIP Congress 74, North Holland Publishing Co., 1974.

Kow79 Kowalski R. A. 'Logic for Problem Solving'. North Holland Publishing Co., 1979.

Mag79 Mago G. A. A Network of Microprocessors to Execute Reduction Languages. Int. Journal of Computer and Information Sciences, Vol. 8 No. 5 and Vol. 8 No. 6, 1979.

Man74 Manna Z. 'Mathematical Theory of Computation'. McGraw-Hill, 1974.

Myc80 Mycroft A. Theory and Practice of Transforming Call By Need Into Call By Value. Fourth Int. Symp. on Programming, 1980.

Pat79 Patel J. H. Processor-Memory Interconnections for Multiprocessors. Proc. Sixth Int. Symp. on Computer Architecture, 1979.

Ree81 Reeve M. J. The ALICE Compiler Target Language. Internal Report, Dept. of Computing, Imperial College, London, 1981.

SlB80 Sleep M. R. and Burton F. W. Towards a Zero Assignment Parallel Processor. Proc. Second Int. Conference on Distributed Computing Systems, 1980.

SFS77 Swan R. S., Fuller S. H. and Siewiorek D. P. Cm*: A Modular Multi-Microprocessor. AFIPS Conf. Proc. Vol. 46, National Computer Conference, 1977.

Tre80a Treleaven P. C. et al. Data Driven and Demand Driven Computer Architecture. Computing Laboratory, University of Newcastle upon Tyne, Internal Report ARM/15, 1980.

Tre80b Proc. of the Joint SRC / University of Newcastle upon Tyne Workshop on VLSI : Machine Architecture and Very High Level Languages. ed. Treleaven P. C. University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, No. 156, 1980.

Tur79 Turner D. A. A New Implementation Technique for Applicative Languages. Software, Practice and Experience, Vol. 9, 1979.

Tur80 Turner D. A. Programming Languages - Current and Future Developments. Proc. of Infotech State of the Art Conference, Software Development Techniques, London, 1980.

Tur81 Turner D. A. Aspects of the Implementation of Programming Languages. D.Phil. Thesis, University of Oxford, 1981.