A Bidirectional Data Driven Lisp Engine for the Direct Execution of Lisp in Parallel

C K Yuen and W F Wong

Dept of Information Systems and Computer Science
National University of Singapore
Kent Ridge, Singapore 0511
(email : wongwf@nusdiscs.bitnet)

1. Introduction

The Bidirectional Data Driven Lisp Engine (BIDDLE) project is an attempt to design an direct execution architecture that will execute Lisp while exploiting its inherent parallelism. Although no particular dialect of Lisp which chosen, BIDDLE is designed with the general features of conventional Lisp in mind. We have also chose not to tackle parallel Lisp dialects. Firstly, it would be more challenging. The second, and the more important, reason was that we felt that there was already a large amount of software written in conventional Lisp that just cannot be ignored. Besides parallelism, BIDDLE also attempts to address the issue of side effects, an area we think is neglected (or not handled well) by most demand and data driven architectures. In this article, we hope to provide the reader with a flavour of the way in which BIDDLE works. This exposure is by no means complete. Due to space constraints detailed discussions about these issues will have to be postponed to elsewhere.

The design of BIDDLE arises from the fundamental observation that a typical statement of a functional program may be recursively compiled into a set of dataflow instructions. Take for example the following statement:

(F1 (F2 (F3 D1 D2) D3) (F4 D4 D5))

Assuming that the functions F1 to F4 all correspond to primitive instructions of the computer, then execution should proceed in the following way:

Operands D1 and D2 are given to instruction F3;
Operands D4 and D5 are given to instruction F4;
Output of F3 and Operand D3 are given to F2;
Output of F2 and F4 are given to F1.

In short, the program corresponds to a dataflow program of four instructions, each being "fired" into execution by the arrival of data. The program is compiled into an execution tree of instructions, with lower nodes (child instructions) sending data to higher ones (parent instructions).

Furthermore, there need not be any separation between two distinct phases of compile and execute, because execution of instructions in one branch can start while a more deeply nested branch is still undergoing compilation. Thus, suppose the machine has an Evaluate instruction EVAL, to which we send the earlier shown program, then the first step of compilation produces:

```
F1 - -                          waiting for two operands
EVAL(F2 (F3 D1 D2) D3)
EVAL(F4 D4 D5)
```

The two new EVAL instructions are the child instructions of F1, to which they will send their output. They may be recursively compiled. However, in actual fact, the evaluation of (F4 D4 D5) amounts to the execution of F4 on data D4 and D5, and this can take place while the first EVAL is producing:

```
F2 - D3                         waiting for one operand
EVAL(F3 D1 D2)
```

That is, the program structure permits the execution of multiple instructions and concurrent compilation and execution, assuming the absence of data dependence, using a data driven execution mechanism in which an instruction becomes executable upon the arrival of its source data.

Research on data driven architectures has produced two main architecture types: Dataflow and Reduction. The former implements the "eager" execution mechanism, in which all executable instructions are put into execution; the "lazy" mechanism of the latter, in contrast, will only execute those instructions whose data are called for by other instructions. For the example shown above there is no difference between the two, as there are no conditional statements. However, consider another example, of a Lisp COND statement with *n* Boolean statements guarding *n* functions:

```
(COND (b1) (f1)
      (b2) (f2)
       ...
      (bn) (fn))
```

In "eager" execution, all the 2*n* child statements b1 to bn and f1 to fn will be put into execution, even though at most one f output will be selected. (None will be selected if all the Boolean guards evaluate to False.) This can obviously be very wasteful. However, the "lazy" mechanism is not ideal either. The strict application of the "lazy" principle to the COND statement would mean that b2 will be evaluated only after b1 evaluates to False, and b3 only after both b1 and b2 produce False, and so on. Execution in this case is strictly sequential. Yet, if the system happens to be lightly loaded at the time and free processors are available, they could have been used to evaluate some of the functions in advance and thus reduce overall execution time.

To overcome this eager/lazy dilemma, BIDDLE adopts the technique of attaching a *priority level* to each object instruction, according to the likelihood that its

result will be needed, but priorities are changed dynamically with the results of earlier execution. An instruction becomes executable upon the availability of data; however, the execution dispatcher will queue executable instructions and only send those of the highest priorities to processing elements. If the load is light, then eager execution results since even low priority instructions are put into execution, but when the machine is heavily loaded, lazy execution is enforced as only the very highest priority instructions in a program are executed.

It is necessary to formulate a set of rules to assign and adjust priority levels. Normally, a child instruction inherits the priority of its parent. However, in a COND statement only the Boolean guard b1, which has to be executed first before one knows whether f1 or b2 will be needed, inherits the full parental priority, while f1 and b2 should get a lower level, and f2 and b3 get a lower level still, and so on. Further, if b1 evaluates to True, then the priority of f1 should increase, while that of all other functions in the COND statement should go to 0 (i.e., the results are not needed and they should not be executed). If b1 evaluates to False, the priority of f1 should go to 0, while b2 should inherit the higher priority. And so on.

Now the Boolean guards b and functions f may themselves be non-primitive functions that compile into a tree of instructions. When their priorities change, the change needs to be transmitted forward to all the descendant instructions. It is this observation which led to the idea of bidirectional links between instructions: while the child instructions must send computed data to their parents, parents must send priority change signals to their children. Thus, a BIDDLE instruction would typically carry three pointers: one to its parent and two to its left and right children. This may seem to be very complicated, but then another observation arises: When a parent instruction receives a data token from a child, the child has already executed, and can therefore be erased from the instruction store. It ceases to be necessary to carry the child instruction address, and the space previously used for the address may now be used to store the received data. Thus, in BIDDLE there is no need for a separate data token store: a data item is directly sent to the instruction waiting for it, and if the instruction already has one data item attached to it, the arrival of the second data token makes it executable. Hence, it is removed from the instruction store and sent to the dispatch queue. The location it previously occupied in the instruction store may be released (but not always immediately) to contain new instructions being produced by the concurrent compilation which, as we remember from earlier discussion, can still be occurring in other branches of the execution tree.

Yet another observation follows: In BIDDLE, instructions are generally not reusable. After an execution takes place, the instruction will be *purged* from the instruction store. Similarly, as soon as earlier execution shows that a result is not needed, such as a function guarded by a Boolean expression that evaluates to False, then instructions for producing that result may be purged also. Whereas in conventional computers recursive calls on the same function causes repeated jumps to the same object code, and in most dataflow machines tokens for different iterations, recursions and reentrant calls may be sent to the same instructions, in BIDDLE each recursive call produces a new set of object instructions. This greatly simplifies compilation since it is unnecessary to refer to any context information about whether reusable code exists and where it is.

It might seem that the use of instruction storage space in BIDDLE would be very inefficient. We argue however that this is not necessarily true. First, as instructions are erased after execution, space can be reused for new instructions being compiled. Second, if an object program is meant to be reused, it must be able to cater for all possible cases, even though during each call only some of the execution paths will be invoked, whereas by compiling anew each time, only code needed for the particular case will be generated. Third, the compilation of any EVAL instruction

frequently produces a number of other EVAL instructions guarded by Booleans. Many of these are not actually executed, but get purged when the guards on them or on their ancestors evaluate to False. In other words, the execution tree may be "pruned" while it is still being produced, and the full memory requirement may never be incurred.

What about the cost of compilation? As will be seen in later sections, the process of compiling functional statements into BIDDLE instructions is extremely simple. Further, because the execution tree is being pruned even when parts of it are still being produced, each call on a function does not necessarily mean that all the constructs contained in the function definition will need to be compiled. So again we argue that the wastage caused by the non-reuse of object code may be acceptable.

Further, the non-reuse of object code produces a number of simplifications. There is no need to tag data tokens with loop counts or activation identifiers, which is a considerable overhead in tagged token machines. Compilation is "context independent" in the sense that one need not know whether an earlier compiled copy exists. An instruction is linked to its parent and children, but need not be stored together with other object code from the same source module. Hence, allocating space to object code is extremely simple. With a small number of exceptions, instructions may be individually relocated as long as the pointers in the parent and children linking back to the instruction are updated.

But what about iterations and loops? BIDDLE does not support loops as such. There is no program counter or branch instructions, and no instruction can send data to itself directly or indirectly since each is purged after execution. In general, iteration is achieved by recursion. However, a number of common iterative operations are taken care of within the BIDDLE architecture as follows:

a. Associative multi-operand operations like summation may be evaluated by constructing an execution tree of binary operations, e.g.: (+ a b c d ...) simply compiles into

```
1: (+ a b)   3
2: (+ c d)   3
3: (+ - -)   7        waiting for two operands
4: (+ . .)   6
5: (+ . .)   6
6: (+ - -)   7        waiting for two operands
7: (+ - -)   15       waiting for two operands
...
```

b. Function application to elements of lists, e.g.: (MAPCAR F (a b c ...) is processed by producing a set of EVAL instructions which all send output to an Assemble instruction, which then collects received elements into a result list:

```
1: ASSEM
2: EVAL(F a)        1
3: EVAL(F b)        1
4: EVAL(F c)        1
...
```

c. Element by element operations on iterative data structures, e.g., adding two arrays, are processed as a single instruction. The data structures are stored in an object store, and their *identifiers* are placed into data tokens and sent to the instructions manipulating them. The arrival of the tokens causes the instruction to be removed from the instruction

store into the dispatch queue. If its priority is sufficiently high, it will get sent to a processing element, which then operates on the data structures in the object store. The ID of the result data structure is then placed in an output token and sent to the parent instruction.

The object store also contains Lisp source programs and the lists which the programs process. An EVAL instruction, for example, may be put into execution by the arrival a list address. As mentioned earlier, this execution may result in additional EVAL instructions but perhaps at different priority levels so that they are only taken for execution at a later time. These EVAL instructions will eventually send data for the parent instruction produced by the first EVAL, e.g.,

EVAL(F1 (F2 D2 D3) (F3 (F4 D4 D5) D6)

produces

1 F1 - -	
2 EVAL(F2 D2 D3)	1
3 EVAL(F3 (F4 D4 D5) D6)	1

While the first child EVAL simply requires executing F2 on D2 and D3 and sending the result back to 1, the other EVAL must itself be compiled into

3 F3 - D6	1
4 EVAL(F4 D4 D5)	3

The whole sequence of events would be the following:

a. The Lisp program (F1 (F2 ...) D6) is placed into the object store;

b. The address of the object is attached to an EVAL instruction and the instruction packet sent to the instruction store; note that an instruction would have a parent address (but in this case the address is null as this is the root instruction of the Lisp program), two child/data addresses (in this case only one as EVAL works on only one object, a Lisp program), and an instruction location address (for the moment undefined).

c. A location address is assigned to the instruction by the instruction store manager, but because the instruction is already complete (a one-object instruction EVAL with a given object address), it is not placed into the instruction store; instead, the allocated address is attached to it and the packet is forwarded to the execution dispatch unit;

d. Assuming that its priority is high enough, the EVAL instruction is sent to a processing element for execution;

e. The processing element detaches the location address of the EVAL, re-attaches this to a new instruction (F1 - -) as its location address, and sends this packet to the instruction store. This non-executable instruction is stored at the given address;

f. The processing element establishes two new objects, namely Lisp programs (F2 D2 D3) and (F3 (F4 D4 D5) D6), and attaches their object IDs to two new EVAL instructions. It also attaches the instruction location originally assigned to the first EVAL instruction to the new EVAL instructions as their *parent* address. These EVAL instructions are then sent to the instruction store,

which assigns them two location addresses. These location addresses may then be inserted into their parent instruction as its child addresses;

g. Both child EVAL instructions are executable, and are sent to the dispatch unit; the first one executes and produces an output data token for the parent instruction F1; the second leads to one instruction (F3 - D6) for the instruction store, and one further EVAL, which will subsequently produce data for F3, which becomes executable upon receipt of the token. F3 produces the second data for F1, which executes to complete the processing required.

Fig. 1 outlines the structure of BIDDLE. Note in particular that the processing elements may output both data tokens and instruction packets on the token bus, and both go directly to the instruction store. There is no data token matching unit. A data token causes the child address of an instruction to be overwritten by a data value or an object ID, and if the matching data value/object address is already present, fires up the instruction. Later we shall see that other effect of data tokens need also be dealt with.

2. Processing Conditional Expressions

Given a conditional expression evaluation

```
EVAL(COND  (b1) (f1)
           (b2) (f2)
            ...
           (bn) (fn))
```

What kind of object code would a BIDDLE processing element produce? A simple method would be the following:

```
1: BOR 2 -
2: CG  - -                                  1
   EVAL(b1)                                 2
   EVAL(f1)                                 2
   EVAL(COND (b2) (f2) ... (fn))            1
```

Here CG is the *Conditional Guard* instruction with the following properties: If the left child evaluates to True, then forward right child output to parent, else discard right child result and send No Result to parent. The BOR is the *Biased OR* instruction defined as: If left child sends result, forward to parent; if left child sends No Result,

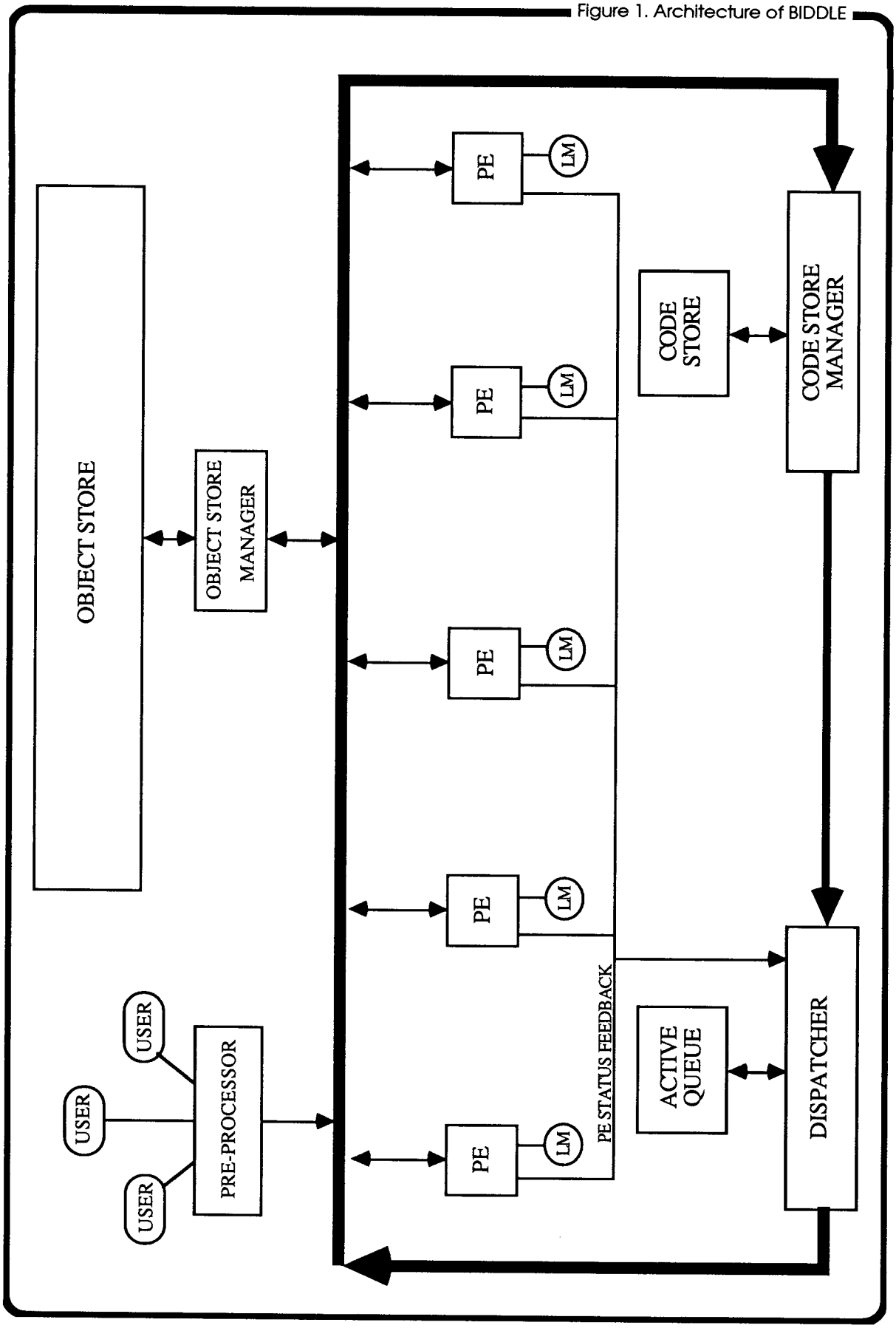

Figure 1. Architecture of BIDDLE

forward right child result to parent. (For the moment we leave from consideration such issues as priorities and purging.)

In short, the processing element would send two instructions into the instruction store, together with three child EVAL instructions which would not be stored because they are already executable. The instruction store manager would merely assign three location addresses to them and send them to the dispatch unit. However, under this method, a long COND statements would cause repeated generations of new EVAL instructions, only for them to travel to the instruction store, get assigned an address, and return for execution. It is obviously more efficient if the original COND gets allocated a whole set of addresses, say from 1 to $2n$, and a single EVAL is executed to produce:

1: BOR 2 5	?
2: CG 3 4	1
3: (b1)	2
4: (f1)	2
5: BOR 6 9	1
6: CG 7 8	5
7: (b2)	6
8: (f2)	6
9: BOR . . .	

(Note that the last number attached to each instruction is the parent address, preceded by two child addresses if known.)

In an execution tree containing BOR and CG, if at the time a CG instruction receives a True result from its left child, the right child has not yet executed, then the priority of the right child and its descendants must be increased. At the same time, the parent of CG, which is BOR, is asked to purge its right branch. When a CG receives a False result from its left child, and if the right child has not yet executed, then the right child and its descendants must be purged. Further, the parent BOR is asked to increase the priority of its right child instruction.

3. Predefined functions

When an EVAL is executed on (F Opr1 Opr2) and F is a primitive operation, a simple object instruction (F - -) results. If however F is itself a function defined elsewhere in the program,

DEF F(x,y) (F1 ...x...y...)

then a new EVAL must be executed on a copy of the definition of F(x,y), with x everywhere replaced by Opr1 and y by Opr2. As mentioned earlier, the Lisp source program is simply an object within the BIDDLE object store, and the definition of F just a component of this object. Similarly, the source statement (F Opr1 Opr2) is a component of the source program object, while Opr1 and Opr2 are even smaller components. EVAL(F Opr1 Opr2) is most easily compiled if the definition of F is preprocessed and represented in the object store as:

(F1 ...^L1...^L2...) (L1,L2)

where ^ denotes a pointer, and L is just a memory location in the object store. The two locations L1/2 attached to the function body constitute the *argument link area*, through which the function body can reach the arguments passed in the function call. Then EVAL(F Opr1 Opr2) is compiled by copying the values of Opr1/2 into L1/2, or placing their addresses there:

EVAL((F1 ...^Opr1...^Opr2...) (Opr1,Opr2), or

EVAL((F1 ...^^Opr1...^^Opr2...) (^Opr1,^Opr2)).

The alternative of searching through the text of F and replacing all occurrences x and y by ^Opr1 and ^Opr2 is more complex and less satisfactory.

If Opr1 is itself a Lisp expression, then the current standard requires it to be evaluated when the function is called. To do that, it is necessary to execute EVAL on the expression and then load its value into the argument link area, i.e., EVAL(F Opr1 Opr2) should copy pointers to Opr1/2 into the link area and produce:

```
EVAL(L1)
EVAL(L2)
EVAL(F ^L1 ^L2)
```

To ensure that the instructions would be executed in sequence, they must be linked together in the following way:

```
1: Forward - -
2: EVAL(L1)                1
3: Forward - -             1
4: EVAL(L2)                3
5: EVAL(F ^L1 ^L2)         3
```

As we shall see later, the data dependence control mechanism of BIDDLE will ensure that any instruction on the left branch would execute before one on the right.

One might ask why we do not adopt the simpler object code

```
1: EVAL(- )
2: MAKELIST(F - -)         1
3: EVAL(Opr1)              2
4: EVAL(Opr2)              2
```

There are two problems. One is that the compilation of F is delayed until after the evaluation of the arguments, whereas in the first arrangement only the execution will be. Second, the evaluated arguments in the first case are linked to the source code of F and will be deleted when the function evaluation completes, but in case two they become "free floating" objects accessed by a number of instructions in function F through the object table but there is no instruction responsible for deleting them.

For functions that define internal variables using the LET statement, e.g.,

DEF F(x,y) LET ((a OprA)(b OprB)...) (...x...y...a...b...),

the convenient way to store the code is to have ^OprA, ^OprB, etc. replacing a,b,etc. inside the LET statement, which acts both as the argument link area and the expression source code, while everywhere else in the function body references to a,b,etc are replaced by pointers to the Opr pointers, e.g., a is replaced by ^^OprA. Executing EVAL on the function causes a set of EVALs to be executed on the LET expressions and one on the function body itself, which gets compiled but will not be executed until after all LET arguments are available.

Note that all the variables x,y,a,b,etc. are statically scoped: they are bound once only, and will not be re-bound to any variables with the same names defined in inner functions.

If the function body and argument expressions only contain primitive operations, then further compilation into BIDDLE object instructions is quite straightforward as shown in Section 1. If however further function definitions need to be invoked to evaluate F1, Opr, etc., then more layers will be added to the pointer structure, with instructions going through several link areas before reaching the object they execute on. Hence the BIDDLE object store need to provide a tagged memory structure so that data values and lists may be distinguished from pointers and pointers to pointers. The processing element will only receive instructions each containing an opcode and two operands, which may be constants, variable data values, object addresses, or starting addresses of pointer chains. The hardware is able to recognize which type of operands it has received, and given a pointer chain, it will follow the chain until a memory location with a non-pointer tag is reached. It will then process the objects.

Further, some pointers only point to components within another object, and since each object may be regarded as its own subset, a second pointer on an object may also be regarded as a component pointer. When the only pointer on an object is erased, the object is also erased, but the erasure of any component pointer will not erase the object itself. Thus, when an object function is executed, the source code for the function should be erased if it is a copy specifically created for this invocation, but if we are merely reusing a previous copy, then the execution only releases the argument link area used by this call.

A number of elaborations are possible:

a. Functional expressions: The function being called may itself be defined by an expression

EVAL((F Opr1 Opr2) Opr3 Opr4),

which is compiled into

EVAL(MAKELIST(EVAL(F Opr1 Opr2) Opr3 Opr4)),

that is, the first EVAL is deferred until the expression has been evaluated and the result re-assembled into a list with the original operands. The list is then given to the original EVAL function to execute on. Here compilation is deferred by necessity, and even the arguments Opr3/4 are not evaluated because we do not know which source code body to link them to.

b. Functional parameters: A function may be passed as a parameter in a call on another function

EVAL(F2 F Opr)

so that, with F2 defined as (F1...x...y...), one would need to execute

EVAL(F1...^^F...^^Opr...).

c. Lambda definitions: A function, including one being passed as a call parameter, may not have a separate definition in the source program; instead, the definition appears as part of the statement being executed, e.g.,

EVAL(Lambda(x,y)(F1...x...y...) Opr1 Opr2)

is supposed to have the same effect as if F1 is a separately defined function. EVAL(F2 Lambda(x,y)(F1...x...y...) Opr) is similarly possible.

The simple way to process this is to store the original Lambda code as

(F1...^^1...^^2...)(^1,^2)

and redirect the EVAL to execute

EVAL((F1...^^Opr1...^^Opr2)(^Opr1,^Opr2)).

In each case, arguments defined by expressions need to be given earlier evaluations as discussed before.

4. Global access

Two instructions can be executed in parallel if they do not supply information to each other, directly or indirectly. If in a BIDDLE program, each instruction only receives information from its children, then there can be no data dependence between instructions in different branches, and all branches can be executed in parallel. Unfortunately this is in general not the case because Lisp statements may refer to environmental variables and objects, both global and local. Since instructions in one branch of the execution tree can change the environmental values used by those in another branch, the latter can execute only after the former have already executed.

BIDDLE considers the whole set of named variables used in a Lisp task as a collection of *Environment Objects*, which are "sent" from instruction to instruction in the precedence sequence specified in the source program. In any execution tree, the left branch has precedence over the right, which gets the entitlement to access the Environment only if the left branch has no need to access it or has already done so. If both branches have no need for access or has already obtained access, the parent instruction will receive the right to access. A single access permission, thus, traverse the execution tree (via the parent/child pointers) in pre-order manner so as to allow instructions to access global and local variables. Due to the many complications, the complete scheme is too lengthy to describe here. The interested reader is referred to our technical report.

5. Conclusion

This article has given a brief introduction to BIDDLE. Our aim here is to argue that the basic principles of BIDDLE are quite straightforward and that we can, indeed,

design an architecture to directly execute Lisp in parallel. As mentioned in the introduction, important and interesting issues like side effect handling, object storage, environment maintenance etc. are not dealt with in great enough details. The interested reader is invited to obtain a copy of the technical report, that is now in preparation, from our department. We also welcome comments and criticisms on these ideas.

References

[1] M. Amamiya, M. Takesue, R. Hasegawa and H. Mikami, "Implementation and Evaluation of A List-Processing-Oriented Data Flow Machine", *Proc. of 13th Int'l Symp. on Comp. Arch.* pp. 10-19. 1986.

[2] D. Hemmendinger, "Lazy Evaluation and Cancellation of Computations", *Proc. of 1985 Int'l Conf. on Parallel Processing*. pp. 840-842. 1985.

[3] J.A. Solworth, "Programming Language Constructs for Highly Parallel Operations on Lists", *J. of Supercomputing*, vol. 2, no. 3, pp. 331-347. Nov 1988

[4] P. C. Treleaven, D.R. Brownbridge and R.P. Hopkins, "Data-driven and Demand-driven Computer Architectures", *ACM Computing Survey*. vol. 14, no. 1, pp. 93-143. Mar 1982.

[5] I. Watson,. V. Woods, P. Watson, R. Banach, M. Greenberg and J. Sargeant, "Flagship : A Parallel Architecture for Declarative Programming", *Proc. of 15th Int'l Symp. on Comp. Arch.* pp.124-130. 1988.

[6] C.K. Yuen and W.F. Wong, "BIDDLE : Preliminary Design of a Bidirectional Data Driven Lisp Engine", *National Univ. of S'pore, Depart. of Info. Sys. and Comp. Sc. Technical Report TRA5/89*. In print.