1991

A Simulation of Demand-Driven Dataflow: Translation from Lucid into MDC Language

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Thomas W. Christopher

A Simulation of Demand-Driven Dataflow: Translation from Lucid into MDC Language

George K. Thiruvathukal
Thomas W. Christopher

Illinois Institute of Technology
Department of Computer Science

Abstract

Message Driven Computation (MDC) is a model of computation with which we have been experimenting at the Illinois Institute of Technology. It is our desire to prove the viability of MDC in practice for the expression of parallel algorithms and the implementation of functional and dataflow programming languages. In the following pages we discuss our implementation of the Lucid programming language in MDC. The discussion will present a subset of Lucid which illustrates the principles of Lucid, Message Driven Computing, and the translation into and the interpretation of dataflow graphs.

1.0 Message Driven Computing

Message Driven Computing (MDC) is a model of parallel and distributed computation developed at the Illinois Institute of Technology by Thomas Christopher **[Christopher 1989]**. Central to MDC is the notion of a computational event. Computational events are executions of functions which map input messages into output messages. All message passing between locations in MDC is achieved unidirectionally and asynchronously. A computational event occurs at a location when a pattern of messages accumulates at the location. Locations are named by computable tuples of information. When two or more computational events occur at a location, mutual exclusion between the computational events is guaranteed. MDC has been implemented on a variety of machines: the Encore Multimax, the BBN Butterfly, and the NCUBE.

2.0 Lucid

Lucid is a family of functional dataflow languages defined and designed by Wadge and Ashcroft **[Wadge and Ashcroft 1985]**. Inherent to the definition of any particular Lucid language are sequences, Lucid operators, pointwise infix and prefix numeric operators, user functions (filters), and list operators (optional).

2.1 Terminology of Lucid

A **sequence** in Lucid is defined to be an infinite series of values ordered (or tagged) by time. The sequence is the basic tenet of Lucid programming. Some examples of sequences are constants, definitions, and the results of function calls. A constant sequence is a sequence whose value at every time is the same. A **definition** of a sequence provides a programmer the facility to have variables which change over time but not to have variables whose history of updates is destroyed. A definition implies that Lucid is a single-assignment language (a tenet of pure dataflow and functional languages). A **function** maps one or more input sequences into an output sequence. In the literature, functions are often alluded to as filters. Sequences are operated on by Lucid operators, pointwise operators, and functions. A Lucid operator is a function which maps one or two input streams onto an output stream whose values are usually values in the history of the input streams.

2.2 Selected Lucid Operators Defined

first

The first operator is applied to a sequence x to produce a constant sequence whose value throughout

TH0363-2/91/0000/0634$01.00 © 1991 IEEE

is the first value of sequence x. Formally, the value of first x at time t is the value of x at time 0.

next

The next operator is applied to a sequence x to produce a sequence in which the value of next x at time t is the value of x at time t+1.

fby

The fby operator is applied to sequences x and y to produce a sequence which is literally the first value of x followed by the sequence y.

2.3 Pointwise Operators

Pointwise operators are operators which are applied to every element of a sequence in much the same manner we are accustomed to in imperative programming languages. Arithmetic operators are all pointwise operators. The operation x + y, applied to sequences x and y, produces a sequence in which each element at a time t is the sum of the value of x at time t and the value of y at time t.

2.4 Language Characteristics

A Lucid program is an expression. Expressions include constants, variables, prefix expressions, infix expressions, list expressions, conditional expressions (if, case, and cond), function calls, and the where clause. A where clause is the mechanism provided for definitions of variables which are bound to expressions. If a variable in an expression cannot be resolved to one of the definitions in its where clause, the variable is a global variable whose history is defined at run time by the user.

3.0 Translation and Interpretation

The system we have developed is a **translator** for Lucid which produces MDC as target code. The translator accepts as input a program expressed in Lucid and produces an MDC initial behavior which constructs a run time dataflow graph whose vertices are MDC **locations** and edges are **messages**. A collation program links the MDC encoded initial behavior with a source file which contains a number of MDC behaviors to interpret the dataflow graph to produce an object MDC program. The MDC program is passed through the MDC translator to produce C code which is finally compiled with UNIX cc and linked with the MDC run time system to produce an executable, parallel program which simulates dataflow on a parallel machine.

3.1 Construction of Dataflow Graphs

We now turn to a concrete example of how our Lucid system constructs dataflow graphs from a simple Lucid source program and the run time behaviors required to completely interpret the dataflow graph. As an aside, we have implemented much more than just a handful of trivial cases, but we believe a simple example best illustrates the principles of translation into and the interpretation of dataflow graphs.

Below is a simple Lucid program which computes the value of x+y+z, given a sequence of natural numbers x, a constant sequence y, and a user-supplied sequence z.

```
x + y + z
  where
       x = 1 fby x + 1;
       y = 1;
    end
```

An initial behavior is produced by the Lucid translator which performs a number of send message operations. Below is MDC pseudocode for the initial behavior which is produced by the Lucid translator. Section A indicates the group of send operations which constructs the dataflow graph for the Lucid program in Figure 1, while section B indicates the group of send operations which establishes a demand pattern at the start vertex (of location) of the dataflow graph for the values of the expression x+y+z. All locations are generated by the translator as needed.

initial behavior
{ Section A }
send a var(x_1) message to location L1.
send a var(y_1) message to location L2.
send a plus(L1,L2) message to location L3.
send a var(z_0) message to location L4.
send a plus(L3,L4) message to location L5.
send a const(1) message to location L6.
send a var(x_1) message to location L7.
send a const(1) message to location L8.
send a plus(L7,L8) message to location L9.
send a fby(L6,L9) message to location x_1.
send a const(1) message to location y_1.

{Section B}
send a demand(global environment, time, destination) message for the first n values of the Lucid program rooted at location L5.
end initial behavior

3.2 Interpretation of Dataflow Graphs

A run time system is required to interpret the MDC dataflow graphs which were produced from the Lucid source program. The run time system is comprised of MDC behaviors. As mentioned earlier in the discussion of MDC, a behavior (computational event) is specified by a pattern of messages and a body of code. The MDC behaviors must be carefully designed to ensure that appropriate message patterns exist for every conceivable dataflow graph produced by the Lucid translator.

We now present a subset of the run time system behaviors. Though it would be interesting to the reader to examine the entire run time system, dozens of pages would be required (merely to present pseudocode). The subset of run time system behaviors which we unveil in the following sections is sufficient to completely interpret the dataflow graph which was constructed by the initial behavior above.

3.2.1 Interpretation of Infix Operators

behavior/message pattern
plus(e1, e2) and demand(env, t, dest)
actions
Create a new location, s, to perform addition.
Send a do_plus(dest) message to location s.
Let L be the left operand portal at location s.
Send a demand(env,t,L) message to e1.
Let R be the right operand portal at location s.
Send a demand(env,t,R) message to e2.
end behavior

behavior/message pattern
do_plus(dest), lopnd(v1), and ropnd(v2)
actions
Send value(v1 + v2) message to dest.
end behavior

3.2.2 Interpretation of Constants

behavior/message pattern
const(v) and demand(env, time, dest)
actions
send value(v) message to dest.
end behavior

3.2.3 Interpretation of Variables

behavior/message pattern
var(name) and demand(env, time, dest)
actions
Send demand_var(dest) message to location named <env, name, time>.
end behavior

one time behavior/message pattern
demand_var(dest)
actions
Send a demand(env, time, here) message to a location whose name is extracted from the present location name. The name of the location to where the demand is sent contains message information pertinent to the definition of the variable name.

Leave the demand_var(dest) message at this location (so this behavior cannot be executed again).
end behavior

behavior/message pattern
demand_var(dest) and value(v)
actions
Send a value(v) message to dest.
Leave the value(v) message at this location, so a subsequent demand will not result in recomputation a variable at the same time in the same environment.
end behavior

3.2.4 Interpretation of a Lucid Operator

behavior/message pattern
send demand(env, time, dest) and fby(e1, e2)
actions
If time is zero, send a demand(env, 0, dest) message to location e1.
Otherwise, send a demand(env, time-1, dest) message to location e2.
end behavior

3.3 A Wave of Computation

As mentioned earlier, the above behaviors comprise only a fraction of the Lucid run time system implemented in MDC. In effect, these behaviors work together to implement the method of eduction proposed in **[Wadge and Ashcroft 1985]**. The interpretation of the dataflow graphs by the MDC system can be viewed as a traveling wave of messages which commences at the initial behavior with demand messages. Demand messages which arrive at a location trigger behaviors which, also, send demand messages. Ultimately, demand messages lead to the computation of actual values which can be returned to the location from where the original demand was made.

4.0 Other Lucid Implementations

An eductive interpreter for the language pLucid was implemented under UNIX in the C programming language at Arizona State University **[Faustini and Wadge 1987]**. This interpreter is perhaps the greatest of the success stories about Lucid implementation. It implements all of the features of the pLucid language described in **[Wadge and Ashcroft 1985]**. The eduction method implemented in the Arizona State pLucid system is exactly the one outlined in **[Wadge and Ashcroft 1985]**.

A translator is described by Pilgram **[Pilgram 1983] [Wadge and Ashcroft 1985]** which translates Lucid into message-passing actors. The method works for a large number of programs; however, it fails to work for a large number of programs. The programs which have failed to be interpreted are those which involve Lucid operators in function calls.

5.0 Significance

We believe our research is significantly different in its focus from the research described above. The pattern matching capabilities of MDC, which are absent in actors languages, are much better suited to the interpretation of graphs. The notion of a location is particularly well-suited for the storage of variable histories.

Another significant aspect of our research is that our work is portable. The MDC system is written in the C programming language. Parallel versions of the system are available for many machines, while sequential versions of the system are available for practically every machine which has a C compiler.

6.0 Future Research with Lucid

We have a least two goals with respect to our Lucid research. Our foremost goal is to implement the entire Arizona pLucid language with some extensions to support arrays. We are exploring the idea of an APL-like array package for MDC into which our Lucid extensions could be translated.

Our other goal is to finish an implementation of Lucid which performs well on all parallel computer architectures, especially distributed architectures. To do so, optimizations will have to be done by the compiler. Our current implementation does limited optimizations, because it is a prototype. We also believe that the addition of arrays to the language will result in improved performance on such architectures, because of studies we have done on the effects of grain size on speed-up and efficiency **[Christopher 1990]**.

7.0 References

[1] E. Ashcroft, *Easyflow Architecture*, Technical Report, Computer Science Laboratory, SRI International 1985.

[2] T. W. Christopher, *Early Experience with Object-Oriented Message Driven Computing*, Frontiers of Massively Parallel Computing 1990, October 1990.

[3] T. W. Christopher, *Exploration of the Limits that Grain Size Imposes on Speed-up and Efficiency of Two Transitive Closure Algorithms*, Fourth Annual Parallel Processing Symposium, Orange County IEEE, April 4-6, 1990.

[4] A. Faustini and W. Wadge, *An Eductive Interpreter for the Language pLucid*, Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, 1987.

[5] P. Pilgram, *Translating Lucid into Message Passing Actors*, Ph.D. Dissertation, University of Warwick, England.

[6] W. Wadge and E. Ashcroft, *Lucid: the Dataflow Programming Language*, Academic Press, Orlando, Florida, 1985.