Compiling Lucid for the oLucid Multiprocessor

D.B. Skillcorn and A.J. Mazeikis.
Department of Computing and Information Science
Queen’s University
Kingston, Ontario
Canada, K7L 3N6

1. Introduction

Interpreters for Lucid have been in existence for some time [2]. However, the performance of these interpreters limits the size of the program that can be effectively executed, as does their I/O structure. We were motivated to develop a system for direct execution of Lucid because of our use of Lucid as a specification language. Specifications of distributed systems are usually large and could not reasonably be executed in an interpreted environment. Also, the existence of a complete execution environment for Lucid means that development from an abstract specification to a concrete specification, and then to an executing prototype can be carried out within the same environment. Hence, properties of the system proven at one level need not be re-proved at the next level of refinement. This makes the use of Lucid for specifying secure systems rather easier. We are also investigating the application of Lucid as a simulation language and this would also require a high performance execution environment.

There are other benefits. Lucid is particularly appropriate as the target language for loosely coupled multiprocessors and we have been exploring many of the interesting issues in the design of such systems along the way.

2. The Target Architecture

Ashcroft has suggested that eduction is the best way to evaluate Lucid and we agree. However, we also think that a loosely coupled architecture is a better target architecture for Lucid in the long run. Loosely coupled systems are much more extensible because the addition of new processors only affects a small number of existing processors. Loosely coupled systems are also less prone to the bottlenecks that occur in accessing shared memory and dynamic switches in tightly coupled systems.

Our abstract target machine consists of a set of processing elements, connected by a static interconnection network. Each processing element must be capable of executing Lucid operators and also of handling the propagation of demands. A moderately conventional ring, as used in many dataflow machines, is appropriate. The interconnection network does not seem to be a major performance determinant, although this has not been rigorously determined. Any static network with valence at least four and small diameter will serve. The abstract machine has been described in more detail in [6].
The eduction evaluation scheme divides operators into two classes: eager and lazy. Lazy operators are evaluated when their values have been demanded and they have been supplied with an appropriate set of operands. This may involve propagating demands to further ‘upstream’ operators in order to obtain those operands. Although it has never been made explicit, it is clear that the evaluation mechanism should be fully lazy, or call-by-need, since no value should ever be computed more than once.

Eager operators are evaluated using essentially the same mechanism, except that a demand is for a contiguous range of values rather than for a single value. The size of the range is called the watermark. This approach was first suggested by Ashcroft and Jagannathan [3]. It amounts to a dataflow evaluation in which speculative work is only done for eager operators and even then is limited by the watermark.

Input and output are handled within the same framework. Inputs are always treated as lazy and hence the demand for a data input results in a demand to the user to input it. Output values appear in any order on the output device.

**Implementation**

The present version of the system is emulated on a network of seventeen Transputers. The Transputer is a fast processor optimized for the execution of small grained processes communicating using synchronous message passing. The message passing abstraction is reading/writing from channels that form 1-to-1 connections between processes. The native language of the Transputer is OCCAM.

Lucid operators are implemented as OCCAM PROCs. Each PROC describes the channels from which its inputs come and on which its output is sent. Code generation from Lucid consists simply of instantiating these PROCs from a library and connecting up the channels to reflect the arcs of the Lucid program (operator net).

Input and output operators in the Lucid program result in PROCs that are connected to channels that, in turn, are connected to the screen and keyboard. A screen manager maintains a slot for each input value and a column for each output value. Input windows are unlocked when input is demanded by the executing program. Users can place the cursor in any unlocked window to enter input. Hence the order of entry of input is as unconstrained as the semantic model requires.

Output values appear in index order in output columns. The order of appearance corresponds only the order of production by the program.

It is our intention to extend the system so as to allow the screen to be a conceptual window on any part of the I/O system. The present scheme allows only a limited number of variables to be used for I/O.

Evaluation in the prototype is purely eager. There is no inherent reason why eduction could not be implemented, since it would require only some rewriting of the OCCAM PROCs, except that there are some problems with handling recursion of user-defined functions and the split operator. We are not, at the moment, sure of the best way to solve these problems.
3. Compilation

Most parts of the Lucid compiler are completely conventional. However, in common with compilers for other functional languages, the Lucid compiler has an optimization phase after semantic analysis. This phase is primarily concerned with order of evaluation which, in this context, means determining which operators should be evaluated eagerly and which should be evaluated lazily. There are three subtasks within this phase: strictness analysis, estimating communication frequencies, and determining memory requirements. These last two tasks annotate the compiled code so that it may be efficiently allocated to processors for execution.

Strictness Analysis

The present version of the compiler uses the rules for Eazyflow evaluation described by Ashcroft and Jagannathan [3] to determine which variables should be evaluated eagerly and which lazily. However, this analysis is not safe in the sense that it may conclude that a variable may be evaluated eagerly when it should be evaluated lazily. This may result in redundant computation.

Two other approaches are being investigated with a view to revising this stage of compilation. One has been suggested by Jagannathan [4] and is a revision of the Eazyflow approach. The other approach is based on work in strictness analysis from the graph reduction community. If we ignore the fact that variables in a Lucid program represent infinite sequences and use this more conventional strictness analysis [5] to decide on evaluation strategy, we get another categorisation of operators. It is guaranteed to be safe but it may be more conservative than Jagannathan's since it may require a complete sequence to be evaluated lazily when some of its elements could have been evaluated eagerly.

Estimating Communication

It is impossible to determine exact communication patterns between operators since doing so is equivalent to solving the halting problem. Hence we must use heuristics. This is done by the compiler in two steps. The compiler first assigns each edge a weight between 0 and 2 representing the amount of data flowing across that edge. A complete infinite sequence is represented by a weight of 1. Hence any edge corresponding to an eagerly evaluated variable is initially assigned a weight of 1 while any edge corresponding to a lazily evaluated variable is assigned a weight of 2 (because an infinite number of demands also pass across it). In the second step information about the types of operators at the head and tail of the variable arc are used to adjust these weights. For instance, the input to a first operator is weighted 0 since only a single value actually traverses the edge. The non-boolean inputs to an if operator can also be reduced, as can the outputs of a partial operator. The actual reduction can be estimated using rules of thumb that are used in the compiler community, although largely undocumented. For instance, the weights of the incoming data arcs to an if operator can be reduced by 50% on the assumption that the boolean input is true of about half the time. It is also possible to consider the form
of the boolean expression to guess this type of information. For example, in

```plaintext
x
where
  n = 1 fby n + 1;
  x = if n < 10 then 1 else 2 fi;
end;
```

it is relatively straightforward to determine that the weight of the first data input to the if should be 0 and the weight of the other 2. This level of sophistication is not implemented in the present compiler.

**Estimating Memory Requirements**

The amount of memory required by an operator can also be estimated by knowing its evaluation type and the evaluation types of its inputs. In this context, memory requirement is primarily for space in the matching store of the processor. A lazy operator all of whose inputs are lazy needs only a small amount of memory for matching. If any of its inputs are eager, it becomes possible for values on that input to pile up waiting to be consumed. The amount of storage required is related to the watermark chosen for the system. For an eager operator, enough memory must be allocated to allow for all of its inputs to pile up to the depth of the watermark.

The only operator requiring special treatment is the split operator. Because of the call-by-need semantics, the split operator having received a demand from one of its downstream neighbours and having satisfied it, must keep a copy of the datum in case it is requested by another of its downstream neighbours. There is no clean way to determine when it is safe to throw such a value away. Hence, the amount of storage required for a split operator is theoretically unbounded.

We are still thinking about the best solution to this problem. Dispensing with call-by-need semantics creates problems with re-executing the eagerly evaluated part of the graph and ultimately can result in users having to enter input values multiple times, so we are not keen on a solution along those lines.

**Code Generation**

The output of the compiler is a set of Lucid operators together with memory requirements and weights for each of the variables. The output is produced in both an abstract form, suitable for post-processing by a variety of other software, and in the form of OCCAM code.

**An Example**

We illustrate the output produced by the compiler given the following program:

```plaintext
x
```
where
\[
x = 1 \text{ fby } x + y;
\]
\[
y = 0 \text{ fby } x;
\]
end;

This produces the intermediate representation

```
intconst IC.1.1 I E 0
intconst IC.0.2 I E 0
+ E 200 Tmp.1 I E 1 x.2 I E 1 y I E 1
fby E 101 x I E 1 IC.1.1 I E 0 Tmp.1 I E 1
fby E 101 y I E 1 IC.0.2 I E 0 x.3 I E 1
split E 100 x.2 I E 1 x.3 I E 1 x I E 1
EOF
```

which is turn is transformed into the OCCAM code

```
intconst(IC.1.1, 1)
intconst(IC.0.2, 0)
plus (Tmp.1, x.2, y)
fby (x, IC.1.1, Tmp.1)
fby (y, IC.0.2, x.3)
split (x.2, x.3, x)
```

4. Partitioning

Before the compiled program can be executed it must be divided into groups of operators that will fit into the processing elements. The weights described in the previous section are used for this purpose.

The cost of communication between two operators in different processing elements is much greater than for similar communication within the same processing element (perhaps as much as two orders of magnitude). Hence a good partition of the program is one that minimizes the weights of the arcs that cross the boundaries of partitions. Of course, the total memory requirements of the operators in a partition cannot exceed the matching memory available in each processing element.

Optimal algorithms to carry out this type of graph partition are known but they have exponential running times (several seconds for graphs of < 10 nodes) and are therefore useless for practical situations. We have carried out an extensive investigation of heuristics, ranging from relatively simple to quite sophisticated. The results of this investigation have been described elsewhere [1].

A package implementing a variety of partitioning algorithms is available as a post-processor for the compiler.
5. Parallel Compilation

Since our target system is itself highly parallel it is productive to consider carrying out the compilation task on the target architecture itself. A great deal of the work in compilation has much inherent parallelism so that the potential gains from this approach are great. There does not seem to have been any concrete investigation of this and so we are currently implementing a system designed to generate some hard data. This will involve rewriting the Lucid compiler as a set of disjoint phases and then implementing each phase in both a serial and parallel form. Data on the stage at which it becomes profitable to compile in parallel will then be available. Intuitively, if parallel compilation is begun early then there is very little information about good ways to divide the program. After each phase the quality of the partitioning decision will increase, but at the expense of more phases executed sequentially.

Suppose that the compilation task is divided into \( P \) phases, each stage taking a representation of the program from the previous stage, transforming it, and producing a new representation for the subsequent phase. Suppose further that each phase takes time \( t_i \) per unit of original program text, and that there are \( N \) possible processors. Let \( s_0 \) be the size of the input and \( s_i \) be the size of the representation produced after phase \( i \). Each \( s_i \) can be expressed as a fraction of \( s_0 \) and this fraction appears to be relatively stable across programs and across languages (this seems to be believed in the compiler community anyway, even though it is undocumented). If we call these fractions \( \alpha_i \) then

\[
\frac{s_i}{s_0} = \alpha_i 
\]

If we consider dividing a particular phase so that it executes concurrently on the \( N \) available processors, then we assume that the time taken per unit of original program text will be \( t_i/N \). There is an associated cost associated with transmitting the program segments to the \( N \) processors and there is an associated cost in all subsequent stages caused by the need to share information in order to create a coherent program structure. Let the amount of information to be communicated be \( p_i \) per input unit. Then the cost of a compilation that executes concurrently from stage \( i \) is governed by four quantities: the cost of executing the sequential stages, the cost of distributing the program before stage \( i \), the cost of executing the concurrent stages, and the costs of communicating during the concurrent stages. If we assume a communication cost per unit of \( c \) then the cost of the sequential part is given by:

\[
\text{Sequential Compilation} = \sum_{j=1}^{i-1} t_j \alpha_{j-1} s_0
\]

The cost of the initial distribution to processors is:

\[
\alpha_i s_0 c
\]

The cost of the concurrent compilation steps is
Concurrent Compilation = \[ \sum_{j=1}^{N} \frac{t_j}{\alpha_{j-1}s_0} \]

Finally the cost of the communication during the concurrent phases is

\[ \text{Communication} = \sum_{j=i}^{P} \alpha_{j-1}s_0 c \]

Our goal is to minimize this quantity with respect to \( i \). For instance, some rearrangement shows that, for sufficiently large \( N \), a reduction in cost is achieved if

\[ p_j c < t_j \]

which captures our intuition that the overall performance is only improved if the cost of connecting program segments is less than the cost of continuing to carry out the compilation serially.

References
4. R. Jagganathan, Private communication (December 1987).