

Adding Eagerness to Eduction*

R. Jagannathan
SRI International
Menlo Park, California 94025
jaggan@csl.sri.com

Abstract

Eduction often unnecessarily constrains evaluation speed to avoid any superfluous evaluation. We describe two simple ways in which the eduction model can be speeded up, through anticipatory and speculative evaluation, both of which avoid unbounded superfluous evaluation. We consider the effects of these extensions on a shared-memory multiprocessor implementation of GLU using idealized programs.

1 Introduction

Eduction is a lazy intensional computing model for evaluating Lucid programs [1, 2]. It is lazy because output values are only produced when they are needed and it is intensional because the individual output values can be produced over time and not necessarily in order. Eduction has been the basis of all known implementations of various incarnations of Lucid (such as pLucid, fLucid, and indexical Lucid) as well as GLU, a hybrid of the latest Lucid and C[3, 2]. Eduction is usually implemented using tagged demand-driven execution and the two are used interchangeably. *Demand* for a value explicitly signifies that the value is needed thus implementing laziness and *tagging* enables individual values to be self identifying thus implementing intensionality.

The main advantage of eduction is that by being lazy it completely avoids unintended or superfluous computations. This is important because a Lucid program conceptually defines infinite universe of values, only a fraction of which is ever needed. Unconstrained eager evaluation, for example, would lead to unbridled superfluous computation making it practically infeasible. The main drawback of eduction as a parallel model of computation is that it is often slower than it needs to be because of its conservative evaluation strategy.

There are two simple ways in which eduction can be made more ‘eager.’

1. The first way is to issue demands for values that would be eventually issued under eduction. This has to be determined typically by a compiler on a per-program basis. It can be easily shown that such anticipatory evaluation does not cause any extra evaluation to occur — only evaluation that would have occurred anyways to occur earlier.
2. The second way is by issuing demands for values that may or may not be demanded by eduction. By evaluating values that possibly may be needed in the future, the total evaluation time can be reduced. However, care has to be taken that the amount of computation due to superfluous demands is not excessive. This is the basis for eazyflow evaluation of temporal Lucid programs [4].

In this paper, we describe ways in which eductive evaluation of multidimensional Lucid and GLU programs can be speeded up. We also measure the effects of these enhancements to eduction in practical terms.

*This work was funded in part by NSF (Grant CCR-9203732).

2 Anticipatory Evaluation

Consider a Lucid program of the form

```
e wvr.a p where
  e = ... f ....;
  f = ... fby.a ... f ...;
  p = ... fby.a ... p ...;
end
```

When `e wvr.a p` is evaluated eductively, values of `p` are evaluated starting from context `#.a = 0`. Whenever the value of `p` at some context is true, the value of `e` at the same context is evaluated. This, in turn, causes the corresponding value of `f` to be evaluated. Since variable `f` is defined in terms of itself, evaluating `f` at a given context would require uncomputed values of `f` at earlier contexts to be evaluated. This would appear to be a major source of slowdown of eductive evaluation.

An obvious solution is to evaluate both the lefthandside and righthandside of `wvr` instead of just evaluating the righthandside. That is, evaluate `e` simultaneously with `p` for increasing contexts. Thus when `p` is true, the corresponding value of `e` would be evaluated without requiring a chain of values of `f` from earlier contexts to be computed.

The main problem with the above solution is that values of `e` are not needed when the corresponding values of `p` are false. Thus, evaluation of such values of `e` is wasteful. (Note however, that all values of `f` are needed since the value of `f` at each context depends on its value at the previous context.)

Instead of evaluating the lefthandside of a `wvr` operator each time the righthandside is evaluated, only those variables that are needed by the lefthandside and that depend on themselves need to be evaluated. In the above example, for each context the value of `p` is evaluated, only the value of `f` at the same context need be evaluated. Whenever, value of `p` at some context is true, value of `e` at that context can be computed from the value of `f` that is being computed from the already-computed value of `f` at the previous context.

This solution avoids any wasteful computation but it requires compile-time dependency analysis of the program to determine when variables used in the lefthandside of `wvr` (or `asa`) can be anticipatorily evaluated.

The effect of anticipatory demands is even more pronounced when dealing with multidimensional computations such as tournament.

Consider the following Lucid program:

```
e asa.a p where
  index a;
  p = ... fby.a ... p ...;
  e = ... fby.a ... F( e @.b 2*(#.b), e @.b (2*(#.b) + 1) ) ...;
end
```

With education, successive values of `p` starting with context `#.a=0` have to be evaluated until `p` is true. Then, `e` is evaluated at that context which causes a tree of demands for values of `e` at earlier contexts to be made. Eventually, the desired value of `e` is produced.

With anticipatory evaluation, we first note that `e` is self-dependent, i.e., each value of `e` determines two values of `e` at the next context. Thus, when `p` at context `#.a=0` is demanded, `e` at the context `#.a=0, #.b=0` is also demanded. Assuming the value of `p` produced is false, the next value of `p` is demanded. At the same time, `e` at context `#.a=1, #.b=0` is demanded which causes `e` at contexts `#.a=0, #.b=0` and `#.a=0, #.b=1` to be demanded. This way, when `p` at context `#.a=t` is demanded, `e` at context `#.a=t, #.b=0` is demanded which results in a tree of values of `e` to be demanded. When `p` at such a context is true, half the values of `e` that determine the value of `e` at that context would have already been demanded.

The impact of anticipatory evaluation on speed is likely to be significant when the time taken to evaluate the righthandside of a `wvr` or `asa` operator is comparable to the time taken to evaluate the lefthandside at a given context.

3 Speculative Evaluation

Consider the following skeletal Lucid program:

```
f wvr.a p where
  f = ... fby.a ...;
  p = ... f ...;
end
```

Notice that p , the righthandside of the `wvr` operator, depends on the variable referred to by the lefthandside, namely f . And the value of variable f at each context depends on its value at the previous context. With eduction, evaluation of the righthandside at each successive context would follow evaluation of the lefthandside (i.e., f) at that context.

Assuming the lefthandside and righthandside take the same amount of time to compute, no advantage is taken of the fact that while p at some context $\#.a$ is being computed from f at the same context, f at the succeeding context, $\#.a+1$, could also be computed at the same time followed by p at the succeeding context.

This is the basis for using speculative evaluation to speed up this kind of program. The basic idea is to evaluate the righthandside of a `wvr` or `asa` operator not only at the current context (as per eduction) but at a bounded number of succeeding contexts at the same time. These succeeding values of the righthandside when needed later would become available much sooner. If these extra values are not needed (such as when the righthandside of `asa` becomes true), then their evaluation would be superfluous. If the computation associated with evaluation of the righthandside at each context is bounded, the total amount of superfluous computation as a result of speculative evaluation is also bounded.

The benefit of speculative evaluation is significant when the computation associated with the righthandside is sizable and when such computation at successive contexts can be overlapped. (Note that anticipatory evaluation would not help speed up the evaluation of this kind of program because the righthandside depends on the lefthandside at each context.)

4 Preliminary Experiments

In order to validate the benefits of the two extensions to eduction, idealized GLU programs that would theoretically benefit from these extensions were developed. These were run using eduction and extended eduction as the models of parallel computation on a multiprocessor and their relative evaluation speed compared.

4.1 Experiment 1

In this experiment, the program shown in Section 2 is evaluated using eduction and using eduction augmented with anticipatory evaluation. For the purposes of the experiment, the time taken to evaluate p ($T(p)$) is the same as the time taken to evaluate f ($T(f)$) which is the same as the time taken to evaluate e ($T(e)$) from f . The rate at which p ($R(p)$) is produced relative to the rate at which e `wvr.p p` ($R(output)$) is produced, is varied from 1 through 25.

Table 1 shows the results of the experiment. When the $R(p):R(e)$ ratio is one, there is no benefit to anticipatory evaluation as every evaluation of the righthandside has a corresponding evaluation of the lefthandside. As this ratio increases, the effect of anticipatory evaluation is evident. For a ratio of 25, the difference between eduction and anticipatory eduction is quite pronounced. It precisely corresponds to the time saved by evaluating 25 values of f simultaneously with 25 values of p .

From this experiment, we can infer that while there is no disadvantage to using anticipatory eduction, it can significantly reduce evaluation speed as when the righthandside of a `wvr` or `asa` expression is infrequently true.

$R(p):R(e)$	eduction	eduction w/anticipation
1	177	177
2	170	163
5	193	172
10	196	164
25	251	176

Table 1: Timings (in seconds): Eduction versus Anticipatory Eduction ($T(p) = T(f) = T(e) = 3s$)

$T(f):T(p)$	eduction	speculation(1)	speculation(2)
1	331	176	177
2	498	338	344
0.5	495	258	167

Table 2: Timings (in seconds): Eduction versus Speculative Eduction

4.2 Experiment 2

In this experiment, the program shown in Section 3 is evaluated using eduction and eduction with speculative evaluation. For the purposes of the experiment, one (speculation(1)) and two (speculation(2)) future values of the righthandside were evaluated speculatively and three different scenarios were considered: $T(f) = T(p) = 3s$, $T(f) = 2T(p) = 6s$, and $2T(f) = T(p) = 3s$.

Table 2 shows the results of the experiment. We can see that when the ratio of $T(f)$ to $T(p)$ is 1, speculation of one future value of the righthandside reduces the evaluation time by approximately half whereas speculation of additional values has no additional benefit. When the ratio of $T(f)$ to $T(p)$ is greater than 1, the reduction in evaluation time is less pronounced and there is no benefit to speculatively evaluation more than one additional value. When the ratio of $T(f)$ to $T(p)$ is less than 1, there is benefit in speculatively evaluation more than one addition value of the righthandside.

We are of course assuming that processors are abundant so that speculatively computing a value does not use up a processing resource needed for more immediate evaluation.

5 Concluding Remarks

We have outlined two techniques for adding eagerness to eduction — anticipatory demands* and speculative demands. Anticipatory demands cause preevaluation of values that *will* be demanded later resulting in possible reduction of evaluation speed. Speculative demands cause preevaluation of values that are *likely* to be needed later in the computation. Through such preevaluation program execution time can often be speeded up considerably. However, a bounded amount of superfluous evaluation can occur.

We have studied these techniques using idealized GLU programs on a multiprocessor and preliminary results suggest that they can significantly speed up certain classes of programs.

Future work will include more detailed study using real applications, incorporation of appropriate dependency analysis into the GLU and Lucid compilers, and suitable modifications to the computing model embodied by the runtime system.

References

- [1] E.A. Ashcroft. Dataflow and eduction: Data-driven and demand-driven distributed computation. Technical Report SRI-CSL-151, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, 1986.

-
- [2] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. Multidimensional Declarative Programming. Oxford University Press, 1994.
 - [3] A.A. Faustini and R. Jagannathan. Multidimensional programming in Lucid. Technical Report SRI-CSL-93-03, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, 1993.
 - [4] R. Jagannathan. A descriptive and prescriptive model for dataflow semantics. Technical Report SRI-CSL-88-5, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, May 1988.