A Linear-System Operator based Scheme for Evaluation of Multinomials

Pavan Adharapurapu and Miloš Ercegovac
University of California, Los Angeles
Computer Science Department
4732 Boelter Hall, Los Angeles, CA 90095, USA
{pavan, milos}@cs.ucla.edu

Abstract

We present a radix-2 online computational scheme for evaluating multinomials in a fixed-point number representation system. Its main advantage is that it can adapt to any evaluation graph representing the multinomial. Evaluation graphs are efficient representations of multinomials in a factored form. The proposed scheme maps subgraphs of the evaluation graph using linear-system operators. These operators transform the expressions represented by the subgraphs into systems of linear equations. The linear equations are then solved in an online, most-significant-digit-first fashion. The scheme produces, after an initial delay, one output digit per iteration for inputs within range. The iteration time is equal to the sum of the delays of a redundant adder, multiplexer, register and a selection unit and is independent of the size of the multinomial and the precision of the inputs/outputs. The initial delay is proportional to the diameter of the evaluation graph and the maximum number of children of any addition node in the graph. The proposed method lends itself to implementation using simple, highly regular hardware with serial interconnections between modules.

1 Introduction

A *multinomial* is a (non-recursive) mathematical expression in several variables consisting of a sum of product terms. Our scheme, however, can also handle the so-called monomials (expressions consisting of just a single product term) and unexpanded multinomials (where each product term could contain some summation raised to a power). Following is an example of a multinomial in three variables which we will use as a running example in this paper.

$$\mathcal{M}(x,y,z) = x + y + \frac{1}{8}x^3y^2 + \frac{1}{16}xyz^4 + \frac{1}{9}x^6yz \quad (1)$$

While such multinomial evaluations are typically done using software on a general-purpose processor, there are situations where custom hardware implementation maybe called for. An example of this — and in fact, the main motivation for the current research — is the Bayesian Network Multinomial (BNM) evaluation. Darwiche [4] recently proposed a scheme where Bayesian Networks [9] are represented using a characteristic multinomial and the most important operation done on a Bayesian Network - *probabilistic inference* - requires evaluation of this multinomial. This example is relevant to our research since BNMs are exponential in size and many real-time Bayesian Networks cannot be practically solved on a general-purpose processor.

One of the chief advantages of the proposed method is that its starting point is any evaluation graph representing the multinomial. An *evaluation graph*, or E-graph for short, is a rooted directed acyclic graph (DAG) whose nodes are represented by the arithmetic operations of addition and multiplication. The leaves of the E-graph are the input variables and the constant coefficients of the multinomial. Symbolic evaluation of the E-graph produces the same expression as the multinomial it represents. E-graphs are efficient representations of multinomials in that they express the multinomial in a factored form.

Define the *size* of an E-graph as the total number of arcs contained in the graph minus the number of nodes in the E-graph. A node of an E-graph with c children represents $(c-1)$ operations to be performed. Thus, the size of an E-graph represents the *total* number of arithmetic operations (of all kinds) to be performed to evaluate the expression represented by the E-graph. A multinomial can have many E-graphs. An E-graph of size 16 for the multinomial (1) is shown in Fig. 1.

The paper does not consider the problem of finding optimal E-graphs for multinomials. Finding small E-graphs for arbitrary multinomials is an open problem and is actively studied in the field of Algebraic Complexity Theory [2, 10]. Finding small E-graphs for BNMs in particular is dealt by Darwiche [4]. Our method, thus, benefits from any advances in these fields.

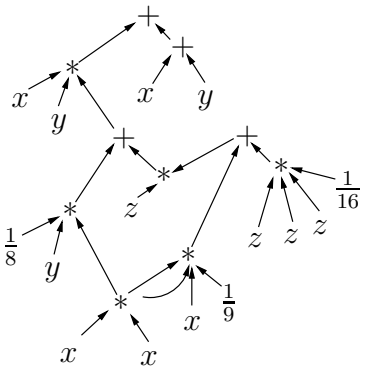

Figure 1. An E-graph of size 16 for $\mathcal{M}$

The method proposed in this paper, named **MOLE** (Multinomial On-Line Evaluation), mechanizes the procedure of building an arithmetic circuit (a hardware implementation consisting of interconnected arithmetic modules) starting from the E-graph of a multinomial. It consists of the following two steps:

1. Application of various linear-system operators to an E-graph representing a multinomial to convert it into a network of linear systems, and
2. Solving of the linear systems in an online fashion with a cycle time independent of the precision [5, 6].

A linear-system operator (LSO) transforms a subgraph of an E-graph into an equivalent linear system. By equivalence, we mean that one of the unknowns of the linear system (typically the unknown whose index is zero) evaluates to the same value (or a power-of-two scaled value) as the expression represented by the subgraph [5].

The rest of the paper is organized in the following way. Section 2 introduces the concept of linear-system operators and the various types of such operators defined for an E-graph. Section 3 is the central part of the paper which details the mapping of an E-graph into a network of linear systems. In section 4, we show when and how scaling of input parameters of a multinomial needs to be performed so that the MOLE method can be applied. In section 5, we compute the delay-area costs of the MOLE method and make comparisons with a conventional approach based on multiply and add units. Section 6 concludes the paper. Appendix A lists the pseudo code used to describe the algorithms developed in the paper.

2 Linear-System Operators (LSO)

Linear-system operators (LSO) map (possibly composite) arithmetic operations to equivalent linear systems. In the context of an E-graph, an LSO maps a subgraph (which is a graph representation of an arithmetic operation) into a linear system. An important requirement while considering an E-graph in this context, is to come up with a suitable set of LSOs for it. An E-graph, as defined previously, contains the operations of multi-operand addition and multi-operand multiplication as its nodes. Thus, LSOs for these two operations seem an obvious choice. In what follows, in addition to above two LSOs, we also define an LSO which maps a composite operation consisting of a combination of additions and multiplications.

2.1 Multiplication LSO (MLSO)

The Multiplication Linear-System Operator (MLSO) maps a multi-operand multiplication operation to an equivalent linear system.

Consider the product p and its corresponding node in an E-graph shown in Fig. 2.

Figure 2. A product and its corresponding node in an E-graph

Now, consider the following linear system $\mathcal{L} : \mathbf{Ay} = \mathbf{b}$:

$$\begin{pmatrix} 1 & -\lambda_0 & 0 \\ 0 & 1 & -\lambda_1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \lambda_2 \end{pmatrix} \tag{2}$$

It is easy to see that solving for y_0 yields p. Thus, the linear system $\mathcal{L}$ is equivalent to the product p.

2.2 Addition LSO (ALSO)

Consider a summation of three terms and its corresponding node in an E-graph shown in Fig. 3.

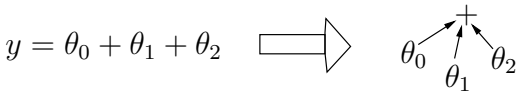

Figure 3. A summation and its corresponding node in an E-graph

It is not possible to map it to a linear system whose value is exactly equal to the sum y and which also satisfies the convergence requirements of the algorithm (presented in

section 2.4) to solve linear systems. We choose to use the following linear system for a sum[1]:

$$\begin{pmatrix} 1 & -\frac{1}{2} & 0 \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \cdot \theta_0 \\ \frac{1}{2} \cdot \theta_1 \\ \theta_2 \end{pmatrix} \quad (3)$$

Solving for y_0 yields $\frac{1}{4} \cdot (\theta_0 + \theta_1 + \theta_2)$ i.e., one-fourth the value of the actual sum. This means that we need to discard the first two most significant digits of the output. It also means that any implementation of the above ALSO needs to run for two extra cycles to produce all the significant digits of the output.

In general, for a summation consisting of n terms, the ALSO produces a value differing from the actual sum by a factor of $2^{-(n-1)}$. Consequently, we need to run the operator for $(n-1)$ cycles more than the required precision.

We will see later that for solving a linear system like (3), all matrix elements should be available at the same time. In the RHS matrix of (3), θ_1 and θ_0 are scaled by $1/2$ and $1/4$, respectively. This means that solving of the linear system (3) can start as soon as θ_2 is available as long as θ_1 and θ_0 are available within one and two additional cycles, respectively. When an ALSO is used as part of a larger E-graph, it takes its inputs from the lower levels. In such a scenario, it is advisable that the sum be mapped so that the elements of the RHS matrix of the ALSO are arranged in decreasing order of production times. This rearrangement is possible since addition is a commutative operation.

2.3 Polynomial LSO (PLSO)

This is the final and the most "powerful" LSO we will define for an E-graph. Unlike the previous two LSOs which deal with individual operations, this LSO deals with a composite operation consisting of a combination of addition and multiplication operations similar to those involved in expressing a polynomial.

Consider the subgraph depicted in Fig. 4. It can, in general, consist of a string of alternating addition and multiplication nodes each having an arbitrary number of children but at the most one parent. Clearly, any similar string of non-alternating addition and multiplication nodes is equivalent to that depicted in Fig. 4 since two successive nodes of the same kind can always be merged. In the figure, the values for all the children of any node are available at the same time.

We define a Polynomial Linear-System Operator (PLSO) for this computation which maps it to a linear system. The PLSO mapping is, in a way, a combination of the MLSO and ALSO. The mapping is presented in (4):

[1]The reader can try other approaches, but the linear system presented here gives the shortest delay.

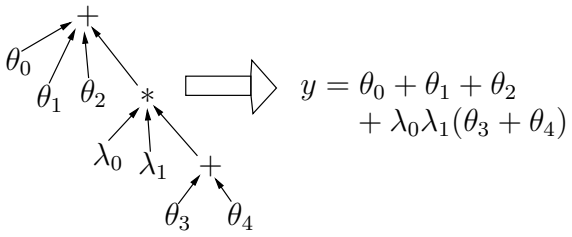

Figure 4. A string of + and * nodes and the expression it represents

$$\begin{pmatrix} 1 & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 1 & -\lambda_0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -\lambda_1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} \frac{1}{8} \cdot \theta_0 \\ \frac{1}{4} \cdot \theta_1 \\ \frac{1}{2} \cdot \theta_2 \\ 0 \\ \frac{1}{2}\,\theta_3 \\ \theta_4 \end{pmatrix} \quad (4)$$

As can be seen from the mapping, we map the multiplication node as in MLSO, except that we leave out one of the operands (λ_0). Similarly, we map the addition node like in ALSO but leave out one of the operands (θ_2). We "link" the two mappings by combining the above left out operands in a single multiply-then-add operation (extract the expression for y_2 in (4)). Note that for mapping the second addition node (the one at the higher level), the RHS matrix elements use different scaling factors than those used in an ALSO. This is because the output of the first addition node (and hence the multiplication node) is already scaled by a certain amount (2^{-1}). We need to take this into account and modify the scaling factors of its RHS matrix elements appropriately. As a side note, the mapping can be varied by taking advantage of the commutative property of addition as explained in section 2.2.

The output produced by a PLSO differs from the actual value by a factor of 2^{-j} where j is the *total* number of children of all the addition nodes minus the number of nodes (both addition and multiplication)[2]. For the above mapping the output produced differs from actual answer by a factor of 2^{-3}.

2.4 Implementing LSOs

We employ the same computational scheme used by the E-method [5] to solve the (linear systems produced by) LSOs in an online fashion [8]. The implementation consists of as many modules as the number of unknowns, each

[2]This value is exact for all but one case; when the root of the string-of-nodes is a multiplication node, j is one more than the above stated value.

computing the value of an unknown y_i given by:

$$y_i = b_i + a_i \cdot y_{i+1} \tag{5}$$

such that one digit of y_i is produced and one digit of y_{i+1} is consumed every cycle, the most significant digits first. Consequently, all equations can be evaluated in parallel and the latency is independent of the number of equations.

The algorithm executed by each of these modules consists of an internal state (called the *residual*) and a recurrence equation to modify the same. For n unknowns forming a linear system $\mathcal{L} : \mathbf{Ay} = \mathbf{b}$, this recurrence is given by (6).

$$w_i[0] = b_i[0] = \sum_{k=0}^{\delta} b_{i,k} 2^{-k};\ a_i[0] = \sum_{k=0}^{\delta} a_{i,k} 2^{-k};\ d_i[0] = 0$$

$$\begin{aligned} w_i[j] = 2(w_i[j-1] + 2^{-\delta-1}\, b_{i,\delta+j} - d_{i,j-1} + \\ a_i[j-1]\, d_{i+1,j-1} + 2^{-\delta-1}\, a_{i,\delta+j}\, D_{i+1}[j-1]), \end{aligned} \tag{6}$$

$$\text{for } \ 0 \le i \le (n-2),\ 1 \le j \le m$$

The equation for the n^{th} module ($i = n-1$) is same as (6) except that the last two addition terms are absent.

The notation used is as follows: the number in square brackets indicates the iteration, the first number in the double subscript is the matrix row index and the second number is the fractional digit position index. In the equation, m is the operand precision and δ is the smallest number of digits of a_i and b_i required to start the algorithm. The minimum value of δ for our implementation is 2. The residual $w_i^{(j)}$ is represented in a redundant form consisting of a pseudo-sum vector WS and a carry vector WC.

In the j^{th} iteration, the i^{th}th module, having calculated the residual $w_i[j]$, selects the output digit $d_{i,j}$ using the following selection function:

$$d_{i,j} = \mathcal{SEL}(\widehat{w_i[j]}) = \begin{cases} 1 & \text{if } \widehat{w_i[j]} \ge 0.5 \\ 0 & \text{if } -0.5 < \widehat{w_i[j]} < 0.5 \\ -1 & \text{if } \widehat{w_i[j]} \le -0.5 \end{cases} \tag{7}$$

where $\widehat{w_i[j]}$ is an estimate of $w_i[j]$ obtained by truncating it to one fractional digit.

The convergence requirement of the algorithm is satisfied if

$$|a_i| \le 1/8, |y_i| \le 1, |b_i| \le 3/4, \quad \text{for all } i \tag{8}$$

For the proof of above equations, see [5] and [7, Chap 10].

Note that when b_i is zero and a_i is non-existent (for equations of the form $y_i = t$), we either don't need any hardware implementation (because it is produced by some other module at a lower level) or we can simply use a shift register (this would be the case when t is a multinomial variable/constant).

Each module requires a [5:2] adder, four registers, two muxes with complementers and a digit selection unit. See [7, page 575] for the implementation details for a similar module. The cycle length for this implementation is estimated to be:

$$t_{cycle} = t_{REG} + t_{MUX} + t_{[5:2]} + t_{sel} \tag{9}$$

indicating that the cycle time is independent of the precision of the operands.

2.5 PLSO vs (MLSO + ALSO)

Consider a string of alternating addition and multiplication nodes that can be mapped using a PLSO. In this subsection, we show why it is better to use a PLSO instead of using a combination of ALSOs and MSLOs.

Let the string have q addition nodes and q multiplication nodes. Let the sum of all addition node children be c. When we use a PLSO to map this string, the delay for the first (correct) output digit to appear is, as described in section 2.3, $(c - 2q + 2)$. The two cycles in this expression is the waiting time for the minimum number of digits (two) of the inputs to arrive before the computation can start.

Now consider the case when we map each individual addition/multiplication node in the string using ALSO/MLSO. For such a mapping, the overall delay (for the production of the first output digit) would consist of the sums of the delays of the individual LSOs. The sum of the delays of all the ALSOs is equal to $(c - q + 2q) = (c + q)$. The delay of each MLSO is equal to two cycles. Thus, the sum of delays of all the LSOs used in the mapping is $(c + 3q)$ which is clearly more than the delay of the PLSO mapping.

3 Applying LSOs to E-graphs

We now describe how to construct an arithmetic circuit for the E-graph representing a multinomial. The approach consists of the following steps. First, we "straighten" out the E-graph into levels and apply the various LSOs to it with an emphasis on PLSOs. Having done that, we determine the start times for the levels.

We need to arrange the E-graph into levels so that we can time the various levels appropriately. This is to make sure that a node doesn't start its computation before the required number of input digits from the lower level nodes are available. We achieve this by arranging the nodes based on their *maximum* distance from the root node. When we arrange the graph in this way (with the root node at the top), then all directed arcs between the different nodes flow

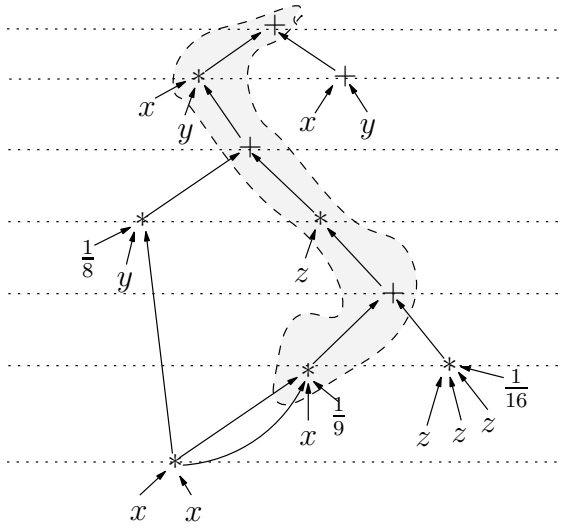

Figure 5. The E-graph of Fig. 1 is arranged into levels. The shaded path indicates the string-of-nodes which are absorbed into a PLSO.

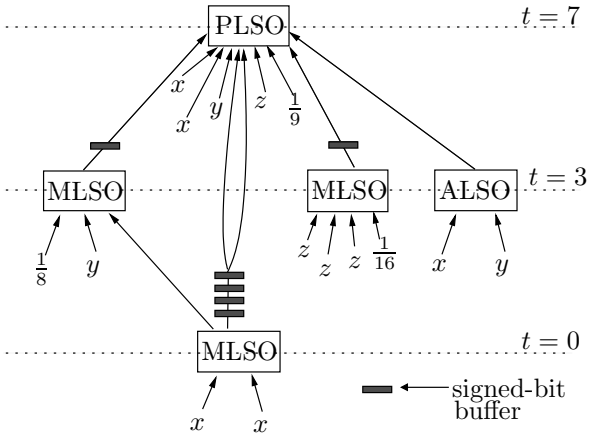

Figure 6. The result of applying LSOs to the "levelled" E-graph of Fig. 5. The starting times are indicated against each level.

from bottom to top. We can find the maximum distance from the root node to all other nodes using the Bellman-Ford single-source longest path algorithm [3]. This algorithm takes $O(r^2)$ time [3], where r is the total number of nodes in the E-graph.

The *diameter* of a graph is defined as the largest path between any two nodes. Since an E-graph is rooted, its diameter path necessarily begins at the root. An E-graph whose diameter is d has d levels when arranged according to the above criteria. The levels increase as we go up, with the root being at level d and the bottom-most level being at level 1. The graph shown in Fig. 1 has a diameter of seven. Fig. 5 shows the "levelled" version of the same graph.

Once we arrange the E-graph in levels, we apply the various LSOs to the nodes or subgraphs making sure we use PLSOs as much as possible. We describe the mapping procedure using C-like pseudo code listed in appendix A. `ApplyLSOs` is the top-level procedure in which we search for occurrences of strings-of-nodes in the E-graph which can be mapped using a PLSO; when we find one, we replace it with a PLSO positioned at the root of the string-of-nodes while making sure all the connections to external nodes are preserved. This mapping could potentially allow us to pull the nodes connected to the string upwards, which is what is done by the `Pull` procedure. Once we exhaust all such strings, we map the rest of the nodes using MLSOs and ALSOs. Fig. 6 shows the result of applying the LSOs to the levelled graph shown in Fig. 5.

The mapping of the E-graph using the various LSOs produces a network of linear systems. Because of the dependencies between the levels, level $(n+1)$ can start executing only after level n starts producing its output digits. If δ_{max} is the maximum online delay of any node in level n, then level $(n+1)$ would have to start as many cycles after level n, or later. Note that the online delay for a module includes the two cycles it has to wait until it consumes the two bootstrapping input digits. An optimization can be done if all the LSOs in level $(n+1)$ are either ALSOs or PLSOs and they were devised to take advantage of the commutative property of addition operation (see section 2.2). In such a case, a given level could be started earlier than the start time calculated above although it is difficult to quantify the savings.

In addition to timing considerations above, we also need to use buffers at the output ports of the LSOs. These are needed to store (some or all) output digits of an LSO when the upper level LSOs are not yet ready to consume them. The size of the buffer required at the output of an LSO is equal to the difference between the time the first (correct) output digit is produced and the latest starting time of any LSO which consumes it, up to a maximum size of m (the working precision).

4 Input Scaling

The E-method works only when the elements of the linear system are bounded (see section 2.4). We discuss in this section how to pre-scale the E-graph inputs so that this requirement is satisfied. These calculations only give us loose scaling factors and can almost certainly be improved for specific E-graphs.

To determine the required scaling we first consider the absolute value of each input at each node. Then, for each level, we determine the largest value emitted by any of the

nodes in that level. In determining this value, we assume that the inputs to this level are all equal to the maximum value emitted by the lower levels. It is easy to see that, for a given level, the maximum value is generated by the addition node with the largest number of children or, in the absence of any addition node, the multiplication node with the least number of children. This is so because the multiplication node's output is less than any of its inputs when all the inputs are less than one (which they will be because of the convergence requirements).

Let the maximum number of children of any addition node in the whole E-graph be u. In the worst case, each level will have one such addition node. Further, set all the input values to the graph as x which will denote the largest input. Using the approach described above, for an E-graph of diameter d the maximum value produced by it (emitted by the root node) is $u^d x$. If we ensure that this value is less than $1/8$ (the smallest of the upper bounds in (8)), then all the elements of the linear systems generated by the E-graph are necessarily less than $1/8$. This is possible if and only if:

$$x < \frac{1}{8u^d} \tag{10}$$

Thus, we need to scale the multinomial so that each and every variable/constant is less than the above number. Define the degree of a term of a multinomial as the sum of powers of each variable in the term, plus one if there is a constant coefficient present. Let the maximum degree of any term in the multinomial be k. Then the scaling factor of the multinomial, denoted by sf, is given by:

$$sf = (8u^d x)^k \tag{11}$$

This is the value by which the given multinomial should be (uniformly) divided before the MOLE method can be applied. If the value of sf comes out less than one, then no scaling is required.

Because of the scaling, the output generated by the arithmetic circuit for an E-graph differs from the actual answer by the same factor. So, we need to run the circuit for an additional number of cycles, denoted by sc (scaling cycles), given by:

$$sc = log_2(sf) \tag{12}$$

This is also the number of initial output digits we need to ignore.

For the E-graph shown in Fig. 5 and for the value $x = 1/512$, the scaling factor and scaling cycles are 2^9 and 9 respectively. These are loose upper bounds and no scaling is, in fact, necessary for this specific E-graph.

5 Area and Delay Complexities

In this section, we calculate the area and delay characteristics of our scheme and compare it with a standard network of conventional arithmetic modules (NCAM). A more detailed comparison of delay/cost for a particular multinomial is presented in [1].

An NCAM implementation consists of replacing each multiplication and addition node in the E-graph with conventional multi-operand multipliers and adders respectively. The inputs/outputs of these conventional modules are digit-parallel. The E-graph will again be arranged in levels based on the maximum distance of each node from the root. Each level will start its execution after its lower levels have finished execution. Since there is no digit-level pipelining involved, the results of a particular node is stored in a buffer whose size is exactly m digits. We might be able to optimize by reusing the adders/multipliers of lower levels, but we will consider the worst case (e.g. there are only one/two levels in the E-graph) and assign each node its own adder/multiplier.

5.1 Area

The hardware for the MOLE method consists of the online modules required for the implementation of the LSOs, together with the buffers required at the outputs of the LSOs.

Consider a multiplication node with v children. The linear system obtained by mapping an MLSO will have v unknowns in it. The implementation of this linear system will require $(v-1)$ online modules since the unknown y_v is either emitted by some other module or can be implemented using a shift register. Similarly, each addition node with v nodes takes $(v-1)$ modules. The modules required for an MLSO are simpler than those required for an ALSO since the linear system for the former has zeroes for all its RHS matrix elements. We will assume, for our analysis, that both MLSO and ALSO use the same module and that this module has a complexity that is the average of the two modules actually required for these two distinct operators. Since a node with v children requires $(v-1)$ modules, the whole E-graph requires as many modules as its size (this assumes the worst case scenario when no PLSOs are applicable). Finally, we need buffers at the output of each LSO. In the worst case, each LSO will need a buffer of size m to store its output.

Consider an E-graph which has r nodes and has a size of $size$. As per the discussion above, the hardware requirements of the MOLE method are:

1. $size$ online modules
2. r buffers of size m each.

We now calculate the hardware requirements of the NCAM implementation. Each multi-operand multiplier (adder) can be built using two-operand multipliers (adders)

composed in a tree-like structure. Clearly, an n operand multiplier (adder) requires $(n-1)$ two-operand multipliers (adders). Thus, for the same E-graph, an NCAM implementation requires:

1. $size$ arithmetic units
2. r buffers of size m each.

Here too, we assume a common arithmetic unit for both addition and multiplication whose complexity is the average of a two-operand adder and a two-operand multiplier. We use a two-operand serial-parallel multiplier and a two-operand carry ripple adder as the basic components for the NCAM implementation.

Plugging in concrete values for the hardware costs for each of the components, we estimate that the common online module uses roughly two times more hardware than the common arithmetic unit. So, in the worst case, the MOLE method is twice as costly as the NCAM implementation. Reference [1] provides a more detailed comparison study.

5.2 Delay

Consider an E-graph of d levels which, in the worst case, has a u-children addition node at each level. Let us suppose that, after the application of the LSOs, the resulting network of linear systems has d' levels with each level having at least one ALSO containing u' inputs. For our analysis, we will assume $ud = u'd'$ and that d' is small compared to d.

As per the earlier discussion on ALSO delay, we need

$$(u' - 1 + 2)d' = u'd' + d' = ud + d' \simeq ud \qquad (13)$$

cycles for the first output digit to be emitted by the root node. In addition, we need to run the circuit for sc cycles extra to compensate for the scaling done to the inputs. Thus, in the worst case, the total cycles required for MOLE method to produce m digits of the output is:

$$T_1 = (log_2((8u^d x)^k) + ud + m) \cdot t_{cycle} \qquad (14)$$

where t_{cycle} is as defined by (9). As (14) shows, for the same inputs, the initial delay is proportional to the diameter of the E-graph and the maximum number of addition-node children. The total delay is also affected by the input sizes. For sufficiently small inputs the scaling cycles tend to be zero.

The E-graph above is also the worst case graph for an NCAM implementation if the u-children addition nodes are replaced by u-children multiplication nodes (since a conventional multiplier has a larger delay than a conventional adder). For such an E-graph, the time taken by the NCAM implementation is simply d (the number of levels) times the delay of the u-operand conventional multiplier. Because of the tree structure, a u-operand multiplier takes $log_2 u$ (the number of tree levels) times the delay of a two-operand multiplier. Let the delay of a two-operand multiplier of precision m be t_m. Then, the total delay of the NCAM implementation is:

$$T_2 = log_2(u) \cdot t_m \cdot d \qquad (15)$$

Note that t_{cycle} is a constant whereas t_m is proportional to m. Plugging in typical component delay values, we estimate that t_{cycle} has a delay of roughly 5 units[3] whereas t_m has a delay of $2.5m$ units. See [1] for the details.

Doing specific calculations for the multinomial (1), we obtain that the first digit for the MOLE implementation is emitted after $11\ t_{cycle}$ or 55 units. The NCAM implementation (which can be visualized by substituting each node in Fig. 5 with the corresponding arithmetic unit), on the other hand, would generate the first (and every other) output digit after roughly $20.5m$ units which for $m = 32$ translates to 656 units. Note that the delay for the MOLE method has a qualitative aspect to it in that it is "pay-per-digit" - we "pay" for only as many digits as are needed.

6 Conclusion

In this paper, we have proposed a linear-system operator based method for evaluating multinomials. It is very practical in nature because it can adapt to any E-graph representation of the multinomial. Although our scheme has greater hardware cost than the conventional scheme, it compares favorably in delay when considering multinomials with small inputs or applications which require only the first few digits of the output. The scheme has the flexible property of allowing as many digits of the output to be produced as needed by the application. Finally the MOLE method, like the E-method on which it is based, can be implemented using simple, problem-independent modules.

7 Future Work

Future work will focus on the following topics:

1. In the current design, all LSOs on a given level start at the same time even if some of them could start earlier. An intelligent global *scheduling algorithm* can be designed for solving this problem. It would strive for the lowest overall delay while also minimizing the buffers between the various LSOs.
2. We need to see if the current design can be optimized when the inputs are discrete (e.g. 0 or 1). Many Bayesian Network E-graphs have discrete variables called *evidence parameters* [4] for many of the inputs.

[3] We use FA delay per bit as the unit delay.

3. We need to investigate if addition of more LSOs (or a completely different set of LSOs) will help us solve the original problem more efficiently.

A Appendix

This appendix lists the algorithms involved in applying LSOs to an E-graph.

```
// Input: d-levelled E-graph
// Output: Network of LSOs
ApplyLSOs(egraph)
{
 currentLevel = d // root
 while(bottom-most level not reached)
 {
  for each node v in currentLevel
  (from left to right)
  {
   nodeStr = GetMaxString(v)
   if Length(nodeStr) > 1
   {
    apply PLSO to nodeStr
    (replace nodeStr with a PLSO at v)
    (preserve external connections)
    Pull(DirectConnectedNodes(nodeStr))
   }
   else if v = Multiplication Node
    apply MLSO
    (replace v with MLSO)
   else if v = Addition Node
    apply ALSO
    (replace v with ALSO)
  }
  currentLevel -= 1
 }
}

// Returns max len string-of-nodes
// starting at v. Each node in the
// string has at most one parent
// Parallel arcs count as 2 parents
GetMaxString(node v)
{
    nodeString = v

    1ParentChilds = all children of v \
     that have only one parent
    if (1ParentChilds == empty)
        return nodeString
    foreach(node w in 1ParentChilds)
        NS[i] = GetMaxString(w)
    nsMax = Max length string in NS
    nodeString += nsMax
    return nodeString
}

// Pulls a set of nodes to higher
// levels when one of the nodes in the
// upper level got absorbed by a PLSO
// and has moved higher
Pull(QueueOfNodes Q)
{
    Sort Q so that higher level \
     nodes are in the front.
    while(Q is not empty)
    {
        u = RemoveFront(Q)
        maxPLevel = Max level of \
         any parent of u
        if(Level(u) < maxPLevel-1)
        {
            Level of u = maxPLevel-1
            Add Children(u) to Q
        }
    }
}
```

References

[1] P. Adharapurapu and M. Ercegovac. A composite arithmetic scheme for the evaluation of multinomials. In *Proc. of the 38th Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 1889–1893, Nov. 2004.

[2] P. Burgisser, M. Clausen, and M. Shokrollahi. *Algebraic Complexity Theory*. Springer-Verlag, January 1997.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge Mass., 1990.

[4] A. Darwiche. A differential approach to inference in Bayesian networks. *Jrnl. of the ACM*, 50(3):280–305, 2003.

[5] M. Ercegovac. A general hardware-oriented method for evaluation of functions and computations in a digital computer. *IEEE Transactions on Computers*, C-26(7):667–680, July 1977.

[6] M. Ercegovac. On-line arithmetic: An overview. In *Real-Time Signal Processing, Proc. SPIE VII*, volume 495, pages 86–93. SPIE, 1984.

[7] M. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, May 2003.

[8] M. Ercegovac, M. Muller, and A. Tisserand. FPGA implementation of polynomial evaluation algorithms. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, pages 177–188, 1995.

[9] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, Sep 1988.

[10] V. Strassen. Algebraic complexity theory. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 633–672. 1990.