A Design of Online Scheme for Evaluation of Multinomials

Pouya Dormiani[a], David Omoto[a], Pavan Adharapurapu[b], and Miloš D. Ercegovac[b]

[a]Electrical Engineering Department, UCLA, Los Angeles, CA 90095, USA
[b]Computer Science Department, UCLA, Los Angeles, CA 90095, USA

ABSTRACT

We present an online arithmetic scheme for hardware evaluation of multinomials arising in Bayesian networks. The design approach consists of representing the multinomial in a factored form as an arithmetic circuit which is then partitioned into subgraphs and mapped to FPGA hardware as a network of online modules connected serially and operating in overlapped manner. This minimizes the interconnect demand without a drastic increase in computation latency. We developed a partitioning/mapping algorithm, designed basic radix-2 online operators and modules, and determined their cost/performance characteristics. We also evaluated the cost/performance characteristics of implementing a Bayesian network on an FPGA chip.

Keywords: Arithmetic circuit, Bayesian network, multinomials, online arithmetic, FPGA implementation.

1. INTRODUCTION

The problem of evaluating multinomials (sums of multivariable products) arises in computing inferences in a Bayesian network (BN). A BN is a compact, graphical model of a probability distribution.[9] It consists of a directed acyclic graph with edges indicating direct causal influences among the nodes (variables) and a set of probability distribution tables quantifying these influences. BNs are widely used in problems where we need to answer questions based on uncertain information. Fig. 1 below shows the well-known ASIA[11] Bayesian network.

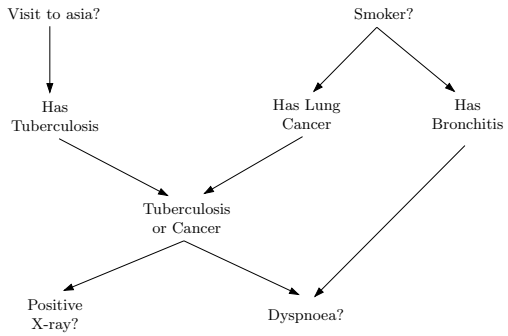


Figure 1. Bayesian network describing the ASIA example.

ASIA depicts the casual structure of a patient having tuberculosis, lung cancer, or bronchitis based on several factors, one being whether or not the patient has recently been to Asia. Probabilistic inference, a common operation performed on BNs, involves finding the probability of an event based on the probability distribution defined by the BN. In the ASIA example, we may want to find the probability of the patient having lung cancer given that he has Dyspnoea.

Darwiche[5] has proposed a new approach for computing inference in BNs wherein he represents the BN using a multinomial and shows that inference can be answered by evaluating the multinomial. To reduce the number of operations, the multinomial is transformed into a factored form which corresponds to a network of multiplication

Further author information: (Send correspondence to M. Ercegovac, milos@cs.ucla.edu)
e-mail: ; P.D.: pouya@seas.ucla.edu; D.O.: domoto@ucla.edu; P.A.: pavan@cs.ucla.edu

Advanced Signal Processing Algorithms, Architectures, and Implementations XV, edited by Franklin T. Luk,
Proc. of SPIE Vol. 5910 (SPIE, Bellingham, WA, 2005) · 0277-786X/05/$15 · doi: 10.1117/12.617342

and addition nodes called the arithmetic circuit[5] (AC) of the Bayesian network. While the multinomial is exponential in size, the corresponding AC is of manageable size. This is achieved by the use of sophisticated algorithms which exploit the global and local structure in the probability distribution.

In this paper, we discuss the design of a hardware arithmetic circuit for evaluating a BN multinomial whose AC representation is given. We consider using online arithmetic[7] to reduce the interconnection requirements of the arithmetic circuit consisting of many operators. A straightforward approach is to map the given AC to a network of online multipliers and adders. Another approach we explored is to use composite online operators, such linear system solvers[4] which, when applicable, have smaller online delay than the delay of online networks. In the next section we discuss obtaining AC forms suitable for implementation. We also discuss organization issues when AC computational requirements exceed available resources. In Section 3 we present the designs and delay/cost characteristics of the online operators: online adder, online multiplier, and linear systems operator (LSO). Section 4 presents estimates of the delay and cost of implementing the ASIA example on a Xilinx FPGA.

2. COMPUTING THE AC

The arithmetic circuit derived from the Bayesian network, as discussed in the last section, is a DAG (directed acyclic graph) in which nodes represent operators or values and edges show dependencies. The inputs of the AC are probabilities, θ, and indicators, λ. The θ values are taken from the probability distribution tables defined for each variable in the Bayesian network, and the λ values specify whether there is evidence that is or is not consistent with a variable.

For our purposes, the AC has a logical, as well as an arithmetic structure. The AC originally derived from the Bayesian network is captured by an NNF (Negation Normal Form) file, in which nodes are represented by a character literal (e.g., "A" for AND), followed by additional parameters that further specify the node. The NNF file currently contains three types of nodes: AND, OR and LEAF nodes. AND nodes are multiplicative nodes; when computing the probability of an event A AND B, the probabilities are multiplied. The OR nodes are similar to AND nodes; when computing the probability of an event A OR B, the probabilities are added. The LEAF nodes contain a probability and/or an indicator. The AND and OR nodes are followed by the number of inputs the operator takes, and a list of indexes that specify the input. Every node has an implicit index according to its position in the file. The very first node in the file has an implicit index of zero. All children of a node will appear in the NNF file before its parent(s). Fig. 2 illustrates this description.

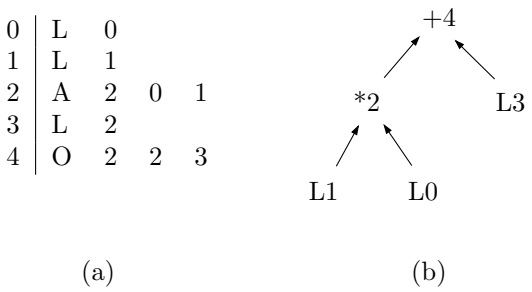


Figure 2. (a) The NNF description (b) The graph representing this description.

Once the structure of the DAG is determined, we must differentiate between arithmetic and logical operators. Some AND nodes will have as inputs, indicators as well as probabilities. Although multiplication and selection by an indicator are logically equivalent, for a hardware implementation we treat selection operations as logical operators and only perform arithmetic when necessary. To determine the nature of an AND operator, we incorporate the LMAP file, which maps values to the leaves of the DAG. Leaves are specified in the NNF as an "L" followed by an index into the LMAP file. The LMAP file contains three types of leaves which are of interest to us: *Indicators* (I), *Probabilities* (P), and *Hybrids* (H) (which act as an *Indicator* AND *Probability*, as shown in Fig. 3c). By recursively examining the inputs to an operator, we can determine whether the operator is of a logic or arithmetic nature. Fig. 3a and Fig. 3b illustrate this concept.

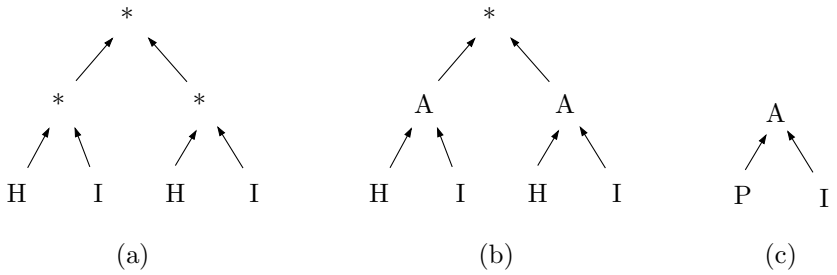


Figure 3. (a) Before LMAP substitution, all operations are considered arithmetic (b) Once the LMAP is substituted, operations that are not strictly arithmetic are changed to logical operations (the nodes labeled "A" are performing logical AND operations and are referred to as L-AND nodes) (c) A hybrid node.

2.1. Partitioning and Mapping

Once an expression has been identified through the NNF and the nature of operations is distinguished with the LMAP file, we can compute inference for the given BN. The process in which the AC can be computed depends on the size of the AC and the hardware resource constraints available. If the AC is small enough to satisfy the hardware constraints, a direct mapping is performed to the operators described in section 3. If the AC demands greater resources than available, it must be computed in smaller iterations. The iterations to make over the AC must take into account dependencies, and sharing (to be discussed in section 2.1.1). Essentially, this involves a two stage process which firstly *partitions* the AC (a DAG) into *subgraphs* which are rooted trees. In order to compute the AC, all subgraphs must be computed in the order intrinsic to their dependencies present in the AC. Certain subgraphs will favor an online approach, and these subgraphs must be *mapped* to the available resources. If a subgraph demands a greater set of resources than available, it must be further *pruned* to create smaller trees which can be mapped to the resources. The prunings in the subgraph will be computed in the order specified by their dependencies. Once all the prunings of a subgraph have been mapped, the subgraph will be computed. This will enable further computation of other dependent subgraphs.

2.1.1. Partitioning

For a given node n in the AC, the *predecessors* (P) of n, act as inputs to n. Also, for a given node n, S is the set of successor nodes which depend on n, then n is a *shared node* if $|S| > 1$. This is shown in Fig. 4.

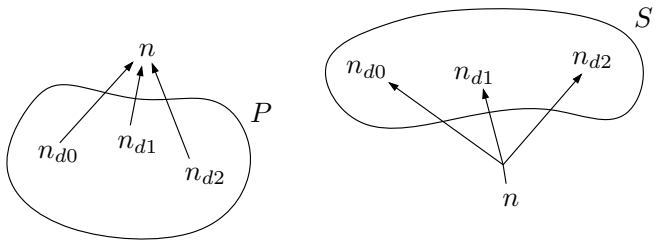


Figure 4. for a given node n, P is the set of predecessors, and S is the set of successors.

Shared nodes contain a value that can be shared anywhere in the network. This means that when computing shared nodes, the output must be saved, so if need be, the value can be used by other iterations. Algorithm 2.1 was used to eliminate shared nodes in the DAG, and create a set of *subgraphs*: rooted trees which contain no shared nodes. The output of a subgraph will be stored and used as inputs to other subgraphs.

Algorithm 2.1: PARTITIONTOSUBGRAPHS(n, V, sg)

```
if n is Operator and n ∉ V
then { P ← PREDECESSOR(n)
       V ← n
       if n is SharedNode
         then { ns ← new Subgraph
                ns ← n
                for each p ∈ P
                  do PARTITIONTOSUBGRAPHS(p, V, ns)
         else { sg ← n
                for each p ∈ P
                  do PARTITIONTOSUBGRAPHS(p, V, sg)
```

PARTITIONTOSUBGRAPHS(n, V, sg) is called with the root of the DAG as n, an empty set containing the visited nodes V, and a newly created root subgraph sg. PREDECESSOR(n) returns a set containing all of node n's predecessors. This algorithm will create a new subgraph at every shared node which is an operator, and place that operator in the newly created subgraph. Then the algorithm recurses on the shared node's predecessors with the newly created subgraph.

Using this algorithm, the PIGS[12] and HAILFINDER[12] arithmetic circuits were partitioned. The charts in Fig. 5 show the distribution of the subgraphs which resulted from the partitioning. The horizontal axis shows the size of the subgraph as the number of operators it contains, and the vertical axis represents the frequency of the given subgraph multiplied by its size. This representation shows the total number of binary operations contributed by a subgraph of a given size.

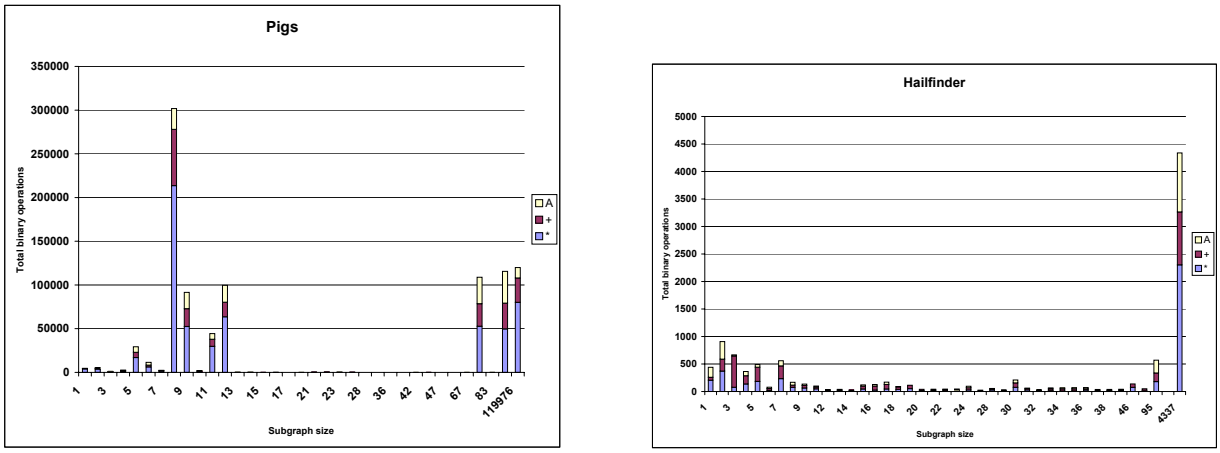


Figure 5. The total number of operators contributed by a given graph size. The distribution of logical nodes (L-AND) and arithmetic nodes is also shown. he size of the PIGS and HAILFINDER networks are approximately 860,000 nodes and 9800 nodes (binary), respectively.

Mapping the AC to an online network requires careful partitioning to expose the characteristics which render online algorithms favorable. The delay through an online chain is characterized by the total online delay through the chain, $\Delta = \sum_i (\delta_i + 1)$, plus the precision of the input, m, which comes about from the serial nature of the algorithm. Long online chains are favorable to amortize the delay contributed by the precision, and conceal the serial delay.

Intuitively, the algorithm is designed to create the largest possible subgraph. This partitioning results in a particular profile in the size of subgraphs created. At one side of the spectrum, are large numbers of small subgraphs, which have no depth, and hence no advantage for online algorithms. At the other side of the spectrum

there are very large subgraphs which are deep and hence amortize the online delay. This profile indicates that both parallel and serial style arithmetic might be suitable for computing the AC. Large subgraphs would be partitioned by pruning the subgraph tree, and mapping it to the resources available. The different iterations required to compute large subgraphs will essentially represent a set of super-operators through which the arithmetic circuit is solved.

2.1.2. Mapping

In the mapping stage, we map a given partition to a set of resources available. At the moment, only the binary operators presented in Section 3 are available. Thus, every operator with a greater number of inputs is restructured as a tree of binary operators. The subgraph is still a rooted tree which makes mapping a somewhat trivial task.

Given a hardware resource constraint describing the availability of operators, we can prune parts of the tree which satisfy the resource constraint. These prunings will create the set of reconfigurations, S, required to fully compute the AC. Each reconfiguration is a super-operator for solving part of the much larger subgraph. If every super-operator, $s_i \in S$, has a given computation time, C_i, and each reconfiguration requires time t_i, then the total time to compute the subgraph is equal to $\sum_{\forall s_i \in S} C_i + t_i$.

One must take into careful consideration that the subgraph still contains L-AND nodes which are logical operations. These operations essentially either feed the value of their child tree to the next dependent node or feed a zero value. These logical operations must be fully exploited to avoid performing computations that become irrelevant due to a L-AND node. This concept will be discussed further in the next section.

2.2. ASIA Example

The ASIA example introduced in Section 1 is used here as a working example to demonstrate how the set of super-operators for the AC are determined. Executing Algorithm 2.1 on the AC derived from compiling the Bayesian network shown in Fig. 1 will result in the subgraphs shown in Fig. 6. As it can be seen, the shared nodes in the AC are the roots of the subgraphs created. From this point on we will only consider the largest subgraph shown, which is the only subgraph created in this example that will benefit from being computed in an online fashion. The subgraphs are subsequently checked for operators with more than two inputs, any such operator is expanded to a tree of binary operators. We now search for "logical and" (L-AND) nodes in the graph by incorporating the LMAP description of the ASIA example. The nodes in the graph which are considered as L-AND operators are marked as "A", shown for the subgraph in Fig. 6.

In our example, the subgraph fits entirely on chip and does not require pruning of the subgraph or reconfiguration. In other words the ASIA example only requires one super-operator, which solves the entire subgraph. Of course for larger graphs this will not be the case and reconfiguration or efficient mapping to a constrained physical routing must be accomplished. At this point in time, we have established that all L-AND nodes should be precomputed for a given subgraph before creating a set of super-operators. The resulting tree created through the precomputation of L-AND nodes avoids any unnecessary computation. We have yet to examine the details of this approach to discover any overhead which this precomputation step may introduce.

3. ONLINE OPERATORS

This section describes the design and implementation of fixed-precision, radix-2 online arithmetic operators and the corresponding modules used in the arithmetic circuit derived from a Bayesian network. For each module, delay and cost as a function of precision are determined. Online addition and online multiplication are presented in Section 3.1 and Section 3.2, respectively. The LSO operators and modules are described in Section 3.3 and Section 3.4, respectively. For all modules, the radix is $r = 2$, m is the precision in bits, δ is the online delay, and t is the truncation precision used for residual estimation. The radix-2 signed-digits for all modules are encoded using the borrow-save notation, where $x \in \{-1, 0, 1\}$ and $x = x^+ - x^-$. For the OLM and LSO modules, the on-the-fly conversion/appending (CA) function is used to convert from the signed-digit representation to the two's-complement representation in a digit-serial manner.[7] The implementations presented are related to those of.[8,10]

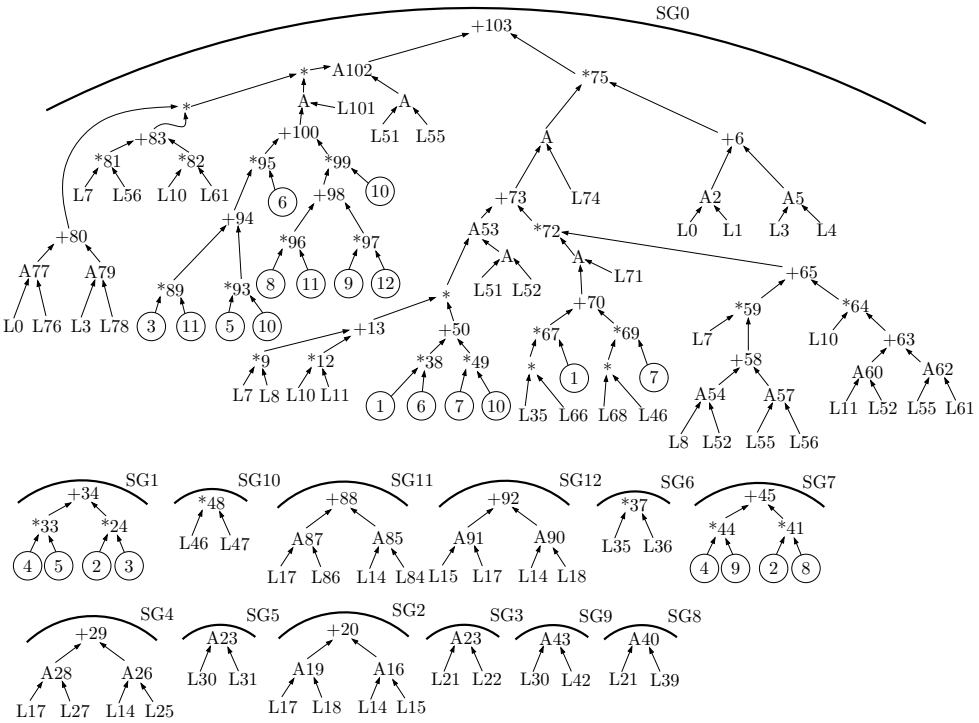


Figure 6. Subgraphs of ASIA. In the subgraphs shown, arithmetic operators are shown by their corresponding operator and L-AND operators are denoted as "A".

3.1. Online Addition

The design and implementation of an online adder (OLA) with $\delta = 2$ will now be discussed. Let $z = x + y$, where x_k and y_k are the digits of the inputs. Fig. 3.1a shows the implementation of the OLA.

The online adder in Fig. 3.1a produces a result in the range [0,1) when $|x|+|y| < 1$, but with $z = z_0.z_1z_2\ldots$. In other words, the first output digit produced has a weight of 2^0. The first digit of x and y consumed by all modules (LSOs, online multipliers, and online adders) are required to have a weight of 2^{-1}. Thus, the online adder must be normalized so its output can be fed into the next module. It can be shown that if $z_0 = 1$, then the next non-zero digit must be -1. Likewise, if $z_0 = -1$, then the next non-zero digit must be 1. Using these facts, the normalization can be done using a simple state machine, as shown in Fig. 3.1b.

The first digit, z_0, produced by the online adder arrives at state $S0$. If this digit is 0, then no normalization is required and edge a is taken to state $S2$. However, if the first digit is non-zero, then edge b is taken to state

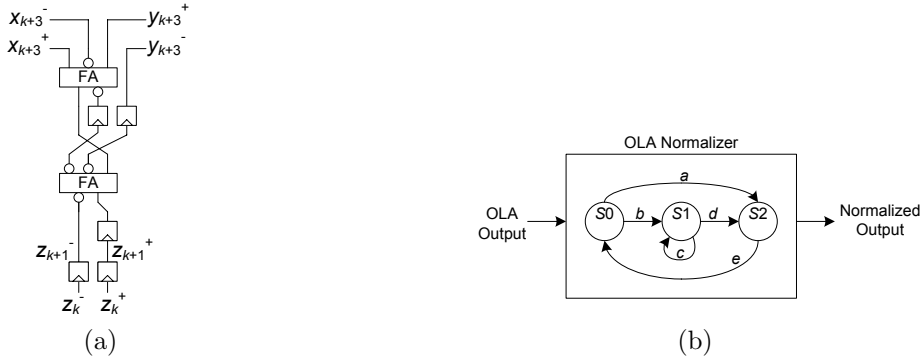


Figure 7. (a) Online Adder, (b) State Machine for OLA Normalizer.

$S1$. A value of 0 is always outputted when the state machine is in state $S0$. The state machine remains in $S1$ (edge c) until a non-zero digit arrives, which then causes the state machine to go to state $S2$ via edge d. While in the $S1$ state, the value of the first digit is outputted. In state $S2$, normalization has completed and the received digit from the online adder is passed through to the output of the normalizer. Edge e is taken once the online adder has produced the m digits, and a new operation can begin.

From Fig. 3.1a, the critical path through the online adder consists of two inverters and two full-adders. To avoid an extra cycle to perform the normalization, z_{j+1} is fed into the normalizer. The normalizer performs an exclusive-or of z^+_{j+1} and z^-_{j+1} to check for a non-zero digit. Thus, the critical path of the online adder and normalizer is equal to Eq. 1. T_{ff} includes the clock-to-q delay from the flip-flops being read as well as the setup time for the flip-flops being written to. The implementation of the online adder with the normalizer in a Xilinx Virtex-4 FPGA requires 22 LUTs and 13 flip-flops, which fits in only 3 CLBs, and can be clocked at over 500 MHz. Note that the implementation remains the same for any m.

$$T_{OLA} = T_{ff} + 2T_{inv} + 2T_{FA} + T_{xor} \tag{1}$$

3.2. Online Multiplication

The design and implementation of an online multiplier (OLM) with $\delta = 3$ and $t = 2$ will now be discussed.[7] Eq. 2 and Eq. 3 show the recurrence equations of the online multiplier. The subscripts denote the digit index and the square brackets indicate the iteration index. Since $\delta = 3$, $j = -3, \ldots, m-1$. Let $x[-3] = y[-3] = w[-3] = 0$. Since the first digit of the product, p_1, is not produced until $j = 0$, the p_{j+1} term is not present in Eq. 3 for $j = -3, -2,$ and -1. Let $x[j] = CA(x[j-1], x_{j+4})$ and $y[j+1] = CA(y[j], y_{j+4})$. The residual, $w[j]$, is in the redundant carry-save form, and is actually represented by two vectors in the two's-complement representation: $ws[j]$ and $wc[j]$. The signed-digits of the product, p_{j+1}, are chosen by the selection function (SEL) using $\hat{v}[j]$, an estimate of $v[j]$ computed with $t = 2$, as shown in Eq. 4.

$$v[j] = 2w[j] + (x[j]y_{j+4} + y[j+1]x_{j+4})2^{-3} \tag{2}$$

$$w[j+1] = v[j] - p_{j+1} \tag{3}$$

$$p_{j+1} = SEL(\hat{v}[j]) = \begin{cases} 1 & \text{if } \hat{v}[j] \geq 0.5 \\ 0 & \text{if } -0.5 \leq \hat{v}[j] < 0.5 \\ -1 & \text{if } \hat{v}[j] < -0.5 \end{cases} \tag{4}$$

Fig. 3.2a shows the possible values of $\hat{v}[j]$ and the corresponding values of p_{j+1} and $\hat{z}[j] = \hat{v}[j] - p_{j+1}$. Let $\hat{v}[j] = \hat{v}^s[j]\hat{v}^0[j]\hat{v}^1[j]\hat{v}^2[j]$ and $\hat{z}[j] = \hat{z}^s[j]\hat{z}^0[j]\hat{z}^1[j]$. The subtraction of p_{j+1} from $\hat{v}[j]$ can be performed using Boolean expressions rather than an explicit subtraction. The corresponding Boolean expressions for p_{j+1} and $\hat{z}[j]$ are shown in Fig. 3.2b.

$\hat{v}[j]$	p_{j+1}	$\hat{z}[j]$
00.0	0	00.0
00.1	1	11.1
01.0	1	00.0
01.1	1	00.1
10.0	-1	11.0
10.1	-1	11.1
11.0	-1	00.0
11.1	0	11.1

(a)

$$\begin{aligned} p^+_{j+1} &= \overline{\hat{v}^s[j]} \cdot (\hat{v}^0[j] + \hat{v}^1[j]) \\ p^-_{j+1} &= \hat{v}^s[j] \cdot (\overline{\hat{v}^0[j]} + \overline{\hat{v}^1[j]}) \\ \hat{z}^s[j] &= \hat{v}^s[j] \cdot \overline{\hat{v}^0[j]} + \hat{v}^s[j] \cdot \hat{v}^1[j] + \overline{\hat{v}^0[j]} \cdot \hat{v}^1[j] \\ \hat{z}^0[j] &= z^s[j] \\ \hat{z}^1[j] &= \hat{v}^1[j] \end{aligned}$$

(b)

Figure 8. (a) Selection of p_{j+1} and calculation of $\hat{z}[j] = \hat{v}[j] - p_{j+1}$, (b) Boolean equations for p_{j+1} and $\hat{z}[j]$.

The five most-significant bits (MSBs) of the datapath, excluding the CA modules, are shown in Fig. 9a. A 4-bit carry-ripple adder (CRA) is needed to compute the estimate $\hat{v}[j]$ with $t = 2$. The selection of p_{j+1} is done

in the SEL block and the calculation of $\hat{z}[j] = \hat{v}[j] - p_{j+1}$ is done in the SUB block. Note that from Fig. 3.2b, the computation of $z[j]$ only depends on $\hat{v}[j]$. Thus, the SEL and the SUB blocks can be performed in parallel. Also note that the terms $x[j]y_{j+4}$ and $y[j+1]x_{j+4}$ from the recurrence equation are multiplied by 2^{-3}. This multiplication is done simply by shifting right by 3 bits. Fig. 9b shows the bit-slice for the repeated bits (RBs). For m bits of precision, there are $(m-3-1)$ instantiations of the RB slice. Fig. 9c shows the least-significant bit (LSB) of the datapath. Recall that multiplying an input by -1 in the two's-complement representation can be done simply by negating each bit of the input and adding a logical 1 to the *unit in the last position* (*ulp*). These "'carry-ins"' are accounted for in the LSB slice.

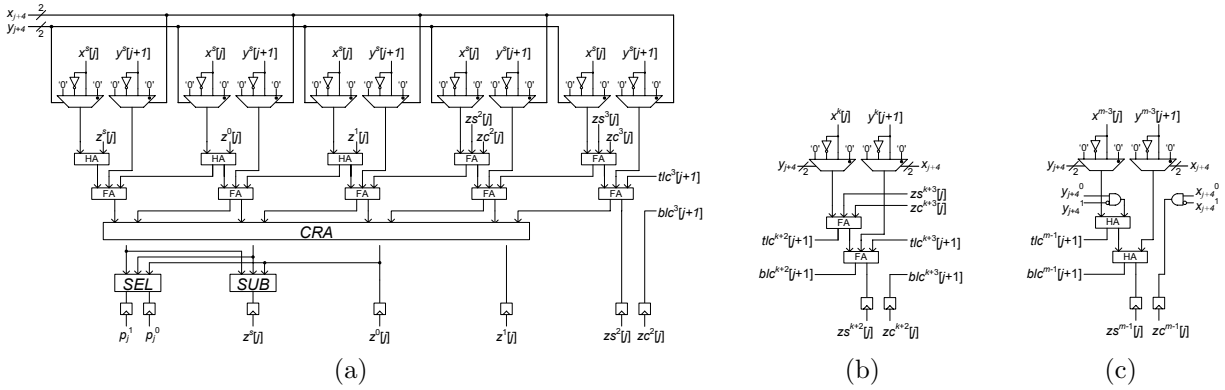


Figure 9. Optimized Datapath for the Online Multiplier: (a) MSBs, (b) RBs, (c) LSB.

$$T_{OLM} = T_{ff} + T_{4\text{-}to\text{-}1\ mux} + T_{[4:2]\ adder} + T_{4\text{-}bit\ CRA} + T_{SEL\ and\ SUB} \tag{5}$$

The critical path of the online multiplier, T_{OLM}, is equal to Eq. 5. T_{ff} takes into account the clock-to-q delay from the flip-flops being read as well as the setup time for the flip-flops being written to. It is assumed that the SEL and SUB functions are performed in parallel and take the same amount of time. This assumption is valid for any 4-input, LUT-based FPGA. In a Xilinx Virtex-4 FPGA, frequencies of over 250 MHz have been achieved. Table 1 shows the LUT and flip-flop usage for several different precisions. The number of LUTs and flip-flops increase linearly with the precision, requiring about 18 LUTs and 8 flip-flops per bit.

m	16	32	48	64
LUTs	278	548	820	1104
flip-flops	129	257	385	513

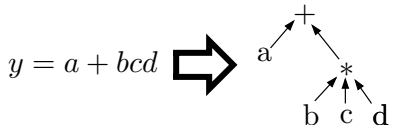
Table 1. Relationship between m and number of LUTs and flip-flops for OLM in Xilinx Virtex-4 FPGAs.

3.3. Linear System Operators

Linear System Operators (LSOs) map an arithmetic expression into an equivalent Linear System of Equations (LS). Equivalence here means that when the LS is solved, one of the variables (typically the "first" variable) evaluates to the same or power-of-two scaled value of the original expression. If we accept that expressions can be represented as E-graphs, an LSO can also be defined as an operator that transforms an E-graph into a LS. Consider the expression and its AC shown in Fig. 10a. The LS produced by applying an LSO to this AC is shown in Fig. 10b. Solving the LS shows that y_0 is indeed equal to the value of the original expression.

Depending on the nature of the AC acted upon, LSOs can be classified into different types. The above LSO is an example of a Polynomial LSO or PLSO because it maps a combination of addition and multiplication operations, similar to a polynomial expression, into a LS. See reference[2] for more details on LSOs.

Mapping arithmetic expressions into LSs by using LSOs is done mainly to take advantage of existing, efficient online algorithms for solving LSs. The scheme proposed in reference[4] solves a LS more efficiently than the



(a)

$$\begin{pmatrix} 1 & -b & 0 \\ 0 & 1 & -c \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a \\ 0 \\ d \end{pmatrix}$$

(b)

Figure 10. (a) An arithmetic expression and its AC, (b) the corresponding LSO.

conventional scheme using a network of online adders and multipliers. However, one restriction in the use of the LSO approach is that inputs should be bounded. While unbounded inputs can be scaled to make them suitable for the LSO method, the extra cycles wasted to "descale" the final results can often neutralize the benefits of this approach. In general, for a linear system $Ay = B$ to be solvable using this scheme, the following convergence requirements[4] should be satisfied: $|a_i| \leq 1/8, |y_i| < 1, |b_i| \leq 3/4$, for all i.

In the context of Bayesian network ACs, LSOs can potentially help us achieve savings in cost/delay if applied carefully. The typical values taken by probability variables (between zero and one) are a bit on the higher side of the convergence upper bounds. Therefore, thoughtless application of LSOs to various portions of the E-graph may not gain anything and may even decrease the efficiency of the overall circuit. But, by careful analysis of the E-graph, regions could possibly be identified where the inputs can be proven to be bounded for all values of leaf input values. Such regions could exist, for example, after a cluster of multiplication nodes. Since probability values are always less than one, the output at each stage diminishes in value as computation propagates up the chain. These optimizations need further research.

3.4. LSO Modules

LSOs can be implemented using variations of the equation, $y_i = b_i + a_i y_{i+1}$, where the subscript i indicates a row in the linear system. An n-row LSO results in $(n-1)$ modules with $\delta = 4$. The n^{th} module $(i = n-1)$ is of the form $y_{n-1} = b_{n-1}$ and can be implemented using just a shift register or can be produced by a different module. The recurrence equation for the i^{th} row is defined in Eq. 6. The square brackets denote the iteration index and the the second subscript denotes the digit index.

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

$$d_{i,j} = SEL(\hat{w}_i[j]) = \begin{cases} 1 & \text{if } \hat{w}_i[j] \geq 0.5 \\ 0 & \text{if } -0.5 < \hat{w}_i[j] < 0.5 \\ -1 & \text{if } \hat{w}_i[j] \leq -0.5 \end{cases} \quad (7)$$

Since $\delta = 4$, j increments from -3 to m. Let $w_i[0] = b_i[0] = 0.b_{i,1}...b_{i,\delta}$, $a_i[0] = 0.a_{i,1}...a_{i,\delta}$, and $d_i[0] = 0$. Eq. 6 and Eq. 7 are executed for $j = 1, ..., m$. Let $a_i[j] = CA(a_i[j-1], a_{i,\delta+j})$ and $d_{i+1}[j-1] = CA(d_{i+1}[j-2], d_{i+1,j-1})$. $\hat{w}_i[j]$ is an estimate of $w_i[j]$ calculated with $t = 3$. The digit-selection function, SEL, chooses the appropriate value of $d_{i,j}$ given $\hat{w}_i[j]$.

The implementation of Eq. 6 will now be discussed. The two's-complement representation is used internally with a carry-save adder (CSA). As a result, the residual, w_i, actually consists of two vectors, ws_i and wc_i. Thus, there are six terms in the recurrence equation. The sum of all the terms is multiplied by 2, which is implemented simply as a left-shift by 1 bit. Let the first two terms of the recurrence equation be defined as $z_i[j-1] = w_i[j-1] - d_{i,j-1}$. Assume $z_i[j-1]$ is calculated in the $(j-1)^{\text{th}}$ cycle, stored in a register, and is available in cycle j. This assumption will be validated later. The third term of the recurrence equation, $2^{-\delta-1} b_{i,\delta+j}$, is simply the value of $b_{i,\delta+j}$ shifted right by $(\delta+1)$ bits. For the fourth term, $d_{i+1,j-1}$ acts as a selector to choose between $-a_i[j-1]$, 0, or $a_i[j-1]$. This can be implemented as a 4-input multiplexer with $d_{i+1,j-1} = 1$ selecting $a_i[j-1]$, $d_{i+1,j-1} = -1$ selecting $-a_i[j-1]$, and 0 otherwise. The same applies to the last term of the recurrence, but now with $a_{i,\delta+j}$ being the selector choosing either $-2^{-\delta-1} d_{i+1}[j-1]$, 0, or $2^{-\delta-1} d_{i+1}[j-1]$.

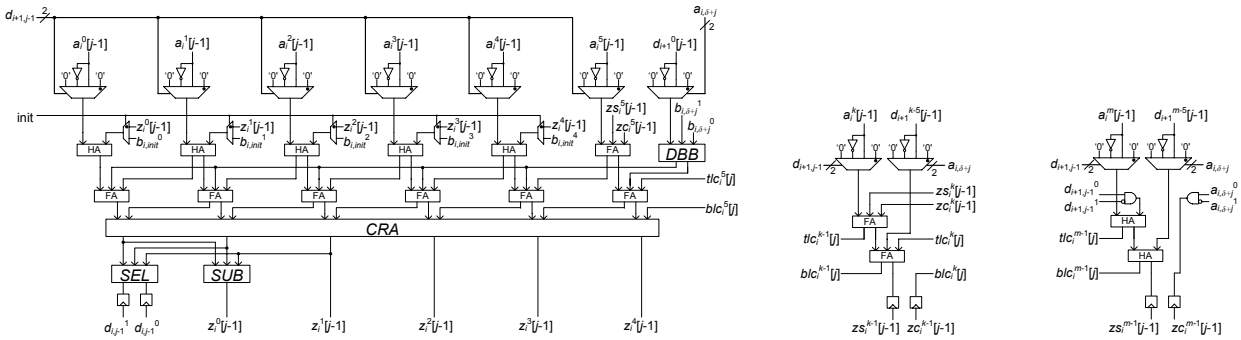


Figure 11. Optimized Datapath for the LSO Module: (a) MSBs, (b) RBs, (c) LSB.

$\hat{w}_i[j]$	$d_{i,j}$	$z_i[j]$
00.0	0	00.0
00.1	1	11.1
01.0	1	00.0
01.1	1	00.1
10.0	-1	11.0
10.1	-1	11.1
11.0	-1	00.0
11.1	0	11.1

(a)

$$
\begin{aligned}
d_{i,j}^{+} &= \overline{\hat{w}^s[j]} \cdot (\hat{w}^0[j] + \hat{w}^1[j]) \\
d_{i,j}^{-} &= \hat{w}^s[j] \cdot (\overline{\hat{w}^0[j]} + \overline{\hat{w}^1[j]}) \\
z_i^s[j] &= \hat{w}_i{}^s[j] \cdot \overline{\hat{w}_i{}^0[j]} + \hat{w}_i{}^s[j] \cdot \hat{w}_i{}^1[j] + \overline{\hat{w}_i{}^0[j]} \cdot \hat{w}_i{}^1[j] \\
z_i^0[j] &= z_i^s[j] \\
z_i^1[j] &= \hat{w}_i{}^1[j]
\end{aligned}
$$

(b)

Figure 12. (a) Selection of $d_{i,j}$ and calculation of $z_i[j] = \hat{w}_i[j] - d_{i,j}$, (b) Boolean equations for $z_i[j]$.

Fig. 11a shows an implementation of the six most-significant bits (MSBs) of the datapath, excluding the CA blocks. Optimizations were made to reduce the number of terms in the recurrence equation so it can be computed using only a [4:2] adder. One optimization allows the computation of $z_i[j-1] = w_i[j-1] - d_{i,j-1}$ to be performed in parallel with the selection function. In other words, $d_{i,j-1}$ need not be explicitly known in order to compute $z_i[j-1]$. The subtraction of $d_{i,j-1}$ from $w_i[j-1]$ for cycle j actually occurs in the $(j-1)^{\text{th}}$ cycle and the result is stored in registers zs_i and zc_i. Fig. 12a shows the selected digit and the computation of $z_i[j]$ for a given $\hat{w}_i[j]$. Note that only one fractional bit of $\hat{w}_i[j]$ is required to select $d_{i,j}$. Let $\hat{w}_i[j] = \hat{w}_i{}^s[j]\hat{w}_i{}^0[j]\hat{w}_i{}^1[j]$ and $z_i[j] = z_i^s[j]z_i^0[j]z_i^1[j]$. Then the Boolean expressions for $d_{i,j}$ and $z_i[j]$ are shown in Fig. 12b, and are displayed in Fig. 11a as the SEL and SUB blocks, respectively. zs_i and zc_i account for two of the inputs into the [4:2] adder.

Another optimization computes $2^{-\delta-1}b_{i,\delta+j} + 2^{-\delta-1}a_{i,\delta+j}d_{i+1}[j-1]$ using Boolean expressions rather than an explicit addition by taking advantage of the fact that the term $2^{-\delta-1}b_{i,\delta+j}$ only requires $(\delta+2)$ bits and the $2^{-\delta-1}a_{i,\delta+j}d_{i+1}[j-1]$ term is sign-extended by $(\delta+1)$ bits. Let B_i be the two's-complement representation of $b_{i,\delta+j}$ and D_{i+1} be the value of $a_{i,\delta+j}d_{i+1}[j-1]$. Fig. 13 shows how the two terms are summed together. Let DBB_i^s be the sign bit and DBB_i^0 be the 5^{th} fractional bit of this combined term. Fig. 14a describes the values of DBB_i^s and DBB_i^0 for every combination of D_{i+1}^s and $b_{i,\delta+j}$. The Boolean expressions for DBB_i^s and DBB_i^0 are given in Fig. 14b, and are represented in Fig. 11a as the DBB block. This combined term becomes the third input of the [4:2] adder. The last input of the adder is the $a_i[j-1]d_{i+1,j-1}$ term.

D_{i+1}^s	D_{i+1}^s	D_{i+1}^s	D_{i+1}^s	D_{i+1}^s	D_{i+1}^s	D_{i+1}^1	D_{i+1}^2	$\cdots$
B_i^s	B_i^s	B_i^s	B_i^s	B_i^s	B_i^0			
DBB_i^s	DBB_i^s	DBB_i^s	DBB_i^s	DBB_i^s	DBB_i^0	D_{i+1}^1	D_{i+1}^2	$\cdots$

Figure 13. Combining the $2^{-\delta-1}b_{i,\delta+j}$ and $2^{-\delta-1}a_{i,\delta+j}d_{i+1}[j-1]$ terms

D^s_{i+1}	$b^+_{i,\delta+j}$	$b^-_{i,\delta+j}$	DBB^s_i	DBB^0_i
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

(a)

$$DBB^s_i = d^s_{i+1} \cdot \overline{b^+_i} + d^s_{i+1} \cdot b^-_i + \overline{b^+_i} \cdot b^-_i$$
$$DBB^0_i = d^s_{i+1} \otimes b^+_i \otimes b^-_i$$

(b)

Figure 14. (a) Selection of DBB^s_i and DBB^0_i, (b) Boolean equations for DBB.

After $w_i[j]$ is computed in carry-save form, a carry-ripple adder (CRA) can be used to compute $\hat{w_i}^3[j]$. The superscript indicates that the $ws_i[j]$ and $wc_i[j]$ vectors are truncated to 3 fractional bits and then added together. Note that in Fig. 11a, $\hat{w_i}^4[j]$ is computed. The reason is that in a Xilinx Virtex FPGA, there is no penalty for computing a 6-bit CRA rather than a 5-bit CRA. Now that $\hat{w_i}[j]$ is known, the SEL and SUB functions can be performed as described previously.

Fig. 11b shows the repeated bits (RBs) of the optimized datapath for the recurrence. For m bits of precision, there are $(m-5-1)$ instantiations of the RB slice. Fig. 11c shows the least-significant bit (LSB) of the datapath. Recall that multiplying an input by -1 in the two's-complement representation can be done simply by negating each bit of the input and adding a logical 1 to the *unit in the last position* (*ulp*). These "'carry-ins"' are accounted for in the LSB slice.

$$T_{LSO} = T_{ff} + T_{4\text{-}to\text{-}1\ mux} + T_{[4:2]\ adder} + T_{6\text{-}bit\ CRA} + T_{SEL\ and\ SUB} \tag{8}$$

The critical path of an LSO module, T_{LSO}, is equal to Eq. 8. T_{ff} takes into account the clock-to-q delay from the flip-flops being read as well as the setup time for the flip=flops being written to. It is assumed that the SEL and SUB functions are performed in parallel and take the same amount of time. In addition, it is assumed that the DBB block has the same delay as a [3:2] adder. These assumptions are valid for an implementation in any 4-input, LUT-based FPGA. The LSO module has a slightly more complex implementation than an online multiplier, resulting in similar delays and resource requirements. In a Xilinx Virtex-4 FPGA, frequencies of over 250 MHz have been achieved. Table 2 shows the LUT and flip-flop usage for several different precisions. The number of LUTs and flip-flops increase linearly with the precision, requiring about 19 LUTs and 10 flip-flops per bit.

m	16	32	48	64
LUTs	318	592	872	1154
flip-flops	157	296	431	597

Table 2. Relationship between m and number of LUTs and flip-flops for LSO Modules in Xilinx Virtex-4 FPGAs.

4. SUMMARY

The AC for the ASIA Bayesian network (Fig. 6) has been implemented in a Xilinx Virtex-4 FPGA using a network of online operators from Section 3. For two-input operators, the original AC without any optimizations requires 55 multipliers and 20 adders. After incorporating the information from the LMAP file, the number of multipliers and adders has been reduced to 29 and 12, repectively. The main cause of the drastic reduction in operators is due to the fact that many AND nodes turn out to be L-AND nodes. The FPGA resource requirements for different values of m, the precision in bits, are given in Table 3. The synthesis and mapping tools were allowed to remove duplicate logic and to replicate logic when needed. Without adding extra latency to the calculation of

m	16	32	48	64
LUTs	6,049	11,545	17,677	23,715
flip-flops	3,391	6,484	9,218	12,591

Table 3. Relationship between m and number of LUTs and flip-flops for ASIA in Xilinx Virtex-4 FPGAs.

the inference, a frequency of 200 MHz has been obtained for all values of m. As evident in Table 3, the number of LUTs and flip-flops required is proportional to the precision.

The network of online adders and online multipliers takes $T_{OL} = 29 + m$ cycles to compute the inference. If LSOs were used in addition to OLAs and OLMs, the time to compute inference becomes $T_{OL+LSO} = 27 + m$. The use of LSOs for this example produces very small savings in cycle time. However, the ASIA arithmetic circuit is so small that the main advantage of LSOs for BNs, the ability to span across multiple levels, is not fully utilized. It is expected that for larger BNs, the advantage of LSOs will be greater. Further research is required to quantify the benefit of LSOs.

The AC for the ASIA Bayesian network is small enough that it can be fully implemented in a single FPGA. For slightly larger ACs, multiple FPGAs can be used. The online scheme is really beneficial in this situation due to the fact that only two wires are needed for a module in one FPGA to communicate with a module in another FPGA. Therefore, the number of I/O pins on the FPGA is usually not the limiting factor in determining how much inter-FPGA module communication there can be, as would be the case if parallel arithmetic was used. For even larger ACs which cannot fit in the available hardware, a new approach must be taken as described in Section 2.2.

Acknowledgements. The authors would like to thank: Mark Chavira for all of his assistance with Bayesian networks and arithmetic circuits, and Deming Chen for providing us with the tools necessary for this research.

REFERENCES

1. P. Adharapurapu and M.D. Ercegovac. "A composite arithmetic scheme for the evaluation of multinomials", *Proc. of the 38th Asilomar Conference on Signals, Systems and Computers*, Vol. 2, pp. 1889–1893, Nov. 2004.
2. P. Adharapurapu and M.D. Ercegovac, "A Linear-System Operator based Scheme for Evaluation of Multinomials", *ARITH-17, 17th IEEE Symposium on Computer Arithmetic*, June 2005.
3. R.H. Brackert, M.D. Ercegovac, and A. Willson. Design of an on-line multiply-add module for recursive digital filters. *Proc. 9th IEEE Symposium on Computer Arithmetic*, pp. 34-41, 1989.
4. M.D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer", *IEEE Transactions on Computers*, vol. C-26, no. 7, July 1977, pp. 667-680.
5. A. Darwiche, "A Differential Approach to Inference in Bayesian Networks", *Journal of the ACM*, vol. 50, no. 3, May 2003, pp. 280-305.
6. M. Ercegovac, J.M. Muller and A. Tisserand, "FPGA Implementation of Polynomial Evaluation Algorithms", *Proc. SPIE*, vol. 2607, FPGAs for Fast Board Development and Reconfigurable Computing, J. Schewel, ed., 1995, pp. 177-188.
7. M.D. Ercegovac, T. Lang: *Digital Arithmetic*, Morgan Kaufmann, 2004.
8. R. Galli and A. F. Tenca, "Design and Evaluation of Online Arithmetic for Signal Processing Applications on FPGAs", *Proc. SPIE*, vol. 4474, pp. 134-44, 2001.
9. J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1988.
10. D. Tullsen and M.D. Ercegovac. Design and implementation of an on-line algorithm. In *Proc. SPIE Conference on Real-Time Signal Processing*, San Diego, August 1986.
11. S.L. Lauritzen, and D.J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems", *Readings in Uncertain Reasoning*, G. Shafer and J. Pearl, Eds. Morgan Kaufmann Publishers, San Francisco, CA, 415-448, 1990.
12. M. Chavira and A. Darwiche, "Compiling Bayesian networks with local structure", *IJCAI-2005.*