Interconnection Scheme for Networks of Online Modules

Pouya Dormiani[a], Miloš D. Ercegovac[b]

[a]Electrical Engineering Department, UCLA, Los Angeles, CA 90025, USA
[b]Computer Science Department, UCLA, Los Angeles, CA 90025, USA

ABSTRACT

In this paper we propose an interconnection scheme to compute any unfactored arithmetic expression as a network of online modules. This is accomplished through mapping the expression to a doubly-linked hypercube network of online units. The mapping algorithm[1] guarantees a maximum dilation of 2, with unit load, and conjectures that any arbitrary unfactored expression can be mapped to the proposed architecture with a small delay overhead. The proposed architecture requires no form of reconfiguration to accomplish the mapping, providing us with an efficient way to compute any network of online operations.

Keywords: Online arithmetic, expression evaluation, interconnection networks

1. INTRODUCTION

Arithmetic expressions are computational graphs with inherent operation dependencies which are known prior to evaluation. Current microprocessor architectures compute expressions as a set of independent operations with intermediate values. This evaluation method has been highly optimized over the years to achieve efficient utilization of the execution unit. The merit of this approach is obviously its generality – the crux of microprocessor architectures. On a processor with multiple execution units, the compiler schedules instructions to obtain the maximum attainable parallelism, which is much less than the available degree of parallelism in large expressions. When evaluating large expressions, many instructions need to be issued at a given cycle, requiring a complex instruction issuing unit, an abundance of execution units, a large register file, and an equally large number of register file read/write ports. As access demands increase, register file performance degrades in-turn degrading instruction performance.

Use of knowledge regarding operation dependencies would help avoid intermediate register accesses by routing data directly from one execution unit to the next. This type of data transfer can quickly become unwieldy even for a small number of execution units when using conventional arithmetic with I/Os consisting of full precision bit vectors. When performing arithmetic in a serial fashion, the number of I/O lines per execution unit are precision independent, and carry a single digit at a time. The reduced I/O demands of serial arithmetic permits an interconnection of a large number of execution units to realize the computational graph of a given expression. However, different expressions have different computational graphs, and require a different interconnection of execution units. The increase in latency due to digit- serial is mitigated by using online arithmetic which allows overlapped execution of subexpressions[2,3] and by using a higher radix. In this paper we present an interconnection scheme for online arithmetic execution units and the associated architecture to compute any unfactored arithmetic expression – an arithmetic expression whose computational graph topology is a tree.

1.1. Online Arithmetic

We now describe a general model of online algorithms and their implementation.[2] Consider an operation with two m radix-r digit operands, x and y, and one result z. The input-output model is:

- In cycle j the result digit z_{j+1} is computed. Consequently the cycles are labeled from $-\delta, \ldots, 0, 1, \ldots, m$ so that in cycle j the operand digits $x_{j+1+\delta}$ and $y_{j+1+\delta}$ are received, output digit z_{j+1} is computed, and output digit z_j is delivered (Figure 1).

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

Advanced Signal Processing Algorithms, Architectures, and Implementations XVI, edited by Franklin T. Luk,
Proc. of SPIE Vol. 6313, 631308, (2006) · 0277-786X/06/$15 · doi: 10.1117/12.680141

The algorithm consists of recurrences on numerical values. In each of the $m + \delta$ iterations, one digit of the operands is introduced (for the last δ iterations the input digits are set to zero), an internal state w (also called a residual) is updated, and one digit of the result is produced (zero for the first δ cycles. An additional cycle is needed to deliver the last result digit.

The principal parts of an online algorithm are

- The recurrence on the residual (internal state) $w[j]$ has the following form

$$w[j+1] = G(w[j], x[j], x_{j+1+\delta}, y[j], y_{j+1+\delta}, z[j], z_{j+1}) \tag{1}$$

  for $-\delta \leq j \leq m-1$ where

$$x[j] = \sum_{i=1}^{j+\delta} x_i r^{-i},\ y[j] = \sum_{i=1}^{j+\delta} y_i r^{-i},\ z[j] = \sum_{i=1}^{j} z_i r^{-i} \tag{2}$$

  are the on-line forms of the operands and the result, respectively.

  The scaled residual is defined as

$$w[j] = r^j(G(f(x[j], y[j]) - z[j])) \tag{3}$$

  with the initial condition $w[-\delta] = 0$.

  A generic form of the recurrence on $w[j]$ is

$$w[j+1] = rw[j] + r^{j+1}(G(f(x[j+1], y[j+1]) - z[j+1]) - G(f(x[j], y[j]) - z[j]))) \tag{4}$$

- The result digit selection obtained as

$$z_{j+1} = F(w[j], x[j], x_{j+1+\delta}, y[j], y_{j+1+\delta}, z[j]) \tag{5}$$

Calling $x[j]$, $y[j]$ and $z[j]$ the numerical values of the corresponding signals when the representation consist of the first $j + \delta$ digits for the operands and j digits for the result, iteration j is described by

$$\begin{array}{lcl} x[j+1] & = & (x[j], x_{j+1+\delta}) \\ y[j+1] & = & (y[j], y_{j+1+\delta}) \\ z_{j+1} & = & F(w[j], x[j], x_{j+1+\delta}, y[j], y_{j+1+\delta}, z[j]) \\ z[j+1] & = & (z[j,], z_{j+1}) \\ w[j+1] & = & G(w[j], x[j], x_{j+1+\delta}, y[j], y_{j+1+\delta}, z[j], z_{j+1}) \end{array} \tag{6}$$

Figure 1 depicts the implementation model.

The execution of an online algorithm corresponds to $m + \delta$ iterations of the recurrence, each corresponding to one clock cycle. The iterations (cycles) are labeled from $-\delta$ to $m - 1$. One digit of each input is introduced during cycles $-\delta$ to $m - 1 - \delta$ and digits value 0 thereafter. The result digits are 0 for cycles $-\delta$ to -1 and z_1 is produced in cycle 0. Finally, the result digit z_j is output in cycle j. Consequently, one additional cycle is required to output z_m.

For an operation with two operands x and y and one output z the execution in cycle j consists of the following actions :

- Input $x_{j+1+\delta}$ and $y_{j+1+\delta}$.
- Update $x[j+1] = (x[j], x_{j+1+\delta})$ and $y[j+1] = (y[j], y_{j+1+\delta})$ by appending the input digits.
- Compute $v[j] = rw[j] + H_1$ ($rw[j]$ is the shifted residual and H_1 is independent of the result digit being produced; its form depends on the operation performed.

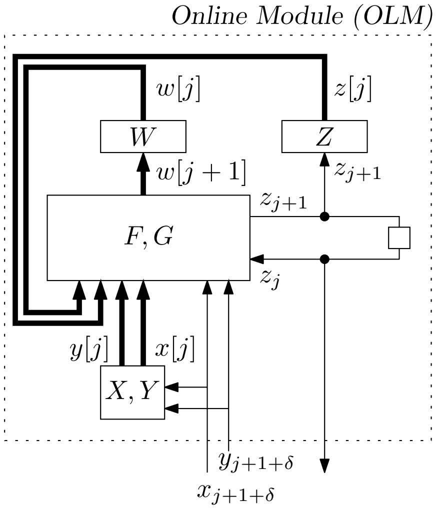


Figure 1. Model of online algorithm implementation (Adapted from [2])

- Determine z_{j+1} using the selection function.
- In some algorithms, update $z[j+1] = (z[j], z_{j+1+\delta})$ by appending the result digits.
- Compute the next residual $w[j+1] = v[j] + H_2(z_{j+1})$
- Digit z_j is output.

The left-to-right mode of computation requires a flexibility in computing output digits on the basis of partial information about inputs. This is achieved by the use of *redundancy* in the number representation system, which allows several representations of a given value.

2. METHODOLOGY

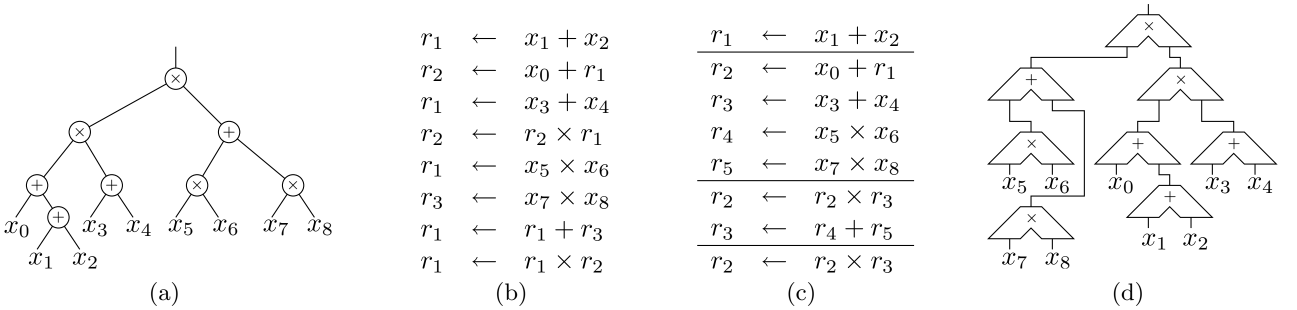
In this section we present our problem statement, justification for its applicability and solution methodology.

Given a set of large expressions which require highly repetitive computation, it is beneficial to optimize the computation of every expression to yield the smallest delay. As the size, number of expressions, and number of times expressions are computed increase, this optimization could yield noteworthy performance improvements. Our objective is to present a methodology which improves the delay of computing large expressions with a high number of repetitions. We consider the case in which expressions have a computational graph with a tree topology.

Consider a set of expressions S_E which comprises all expressions to be computed in a repetitive manner. We assume that the computational graph of any expression $E \in S_E$ has n nodes if the expression contains n operators. If an expression has a structure which is known prior to execution, this information can be used to optimize away access to intermediate values. For example, the computational graph for expression (7) is shown in Fig. 2(a).

$$E = (x_0 + x_1 + x_2)(x_3 + x_4)(x_5 x_6 + x_7 x_8) \tag{7}$$

In a microprocessor approach, this computational graph could be computed so that very few registers are required as shown in Fig. 2(b); however, this approach does not exploit any parallelism. The computation could also be performed to exploit all available parallelism as shown in Fig. 2(c)–instructions grouped together can



$$\begin{aligned}
r_1 &\leftarrow x_1 + x_2\\
r_2 &\leftarrow x_0 + r_1\\
r_1 &\leftarrow x_3 + x_4\\
r_2 &\leftarrow r_2 \times r_1\\
r_1 &\leftarrow x_5 \times x_6\\
r_3 &\leftarrow x_7 \times x_8\\
r_1 &\leftarrow r_1 + r_3\\
r_1 &\leftarrow r_1 \times r_2
\end{aligned}$$

(b)

$$\begin{array}{ccc}
r_1 & \leftarrow & x_1 + x_2\\
\hline
r_2 & \leftarrow & x_0 + r_1\\
r_3 & \leftarrow & x_3 + x_4\\
r_4 & \leftarrow & x_5 \times x_6\\
r_5 & \leftarrow & x_7 \times x_8\\
\hline
r_2 & \leftarrow & r_2 \times r_3\\
r_3 & \leftarrow & r_4 + r_5\\
\hline
r_2 & \leftarrow & r_2 \times r_3
\end{array}$$

(c)

Figure 2. (a) Computational graph for Eq. 7 (b) Instruction sequence to compute the expression requiring very few registers (c) Instruction sequence to achieve maximum available computational parallelism–all instructions grouped together can be issued simultaneously. (d) Interconnection of serial arithmetic operators–all connections in the schmematic are serial.

be issued simultaneously. For large expressions, this approach requires a large number of registers with many read/write ports.

We propose that the computational graph be realized as an interconnection of online arithmetic modules–shown in Fig. 2(d). If $|S_E| = 1$, then the optimum interconnection scheme is the direct connection of operations as specified by dependencies in the computational graph. In this case, since a single large expression is computed repetitively, the interconnection of online modules is static at runtime, and could be optimized for minimum delay. If the expression changes infrequently, then a reconfigurable platform could be utilized to implement the new expression; however, if S_E contains a large number of expressions which need to be computed interchangeably then reconfiguration is far too slow and a more general approach is required.

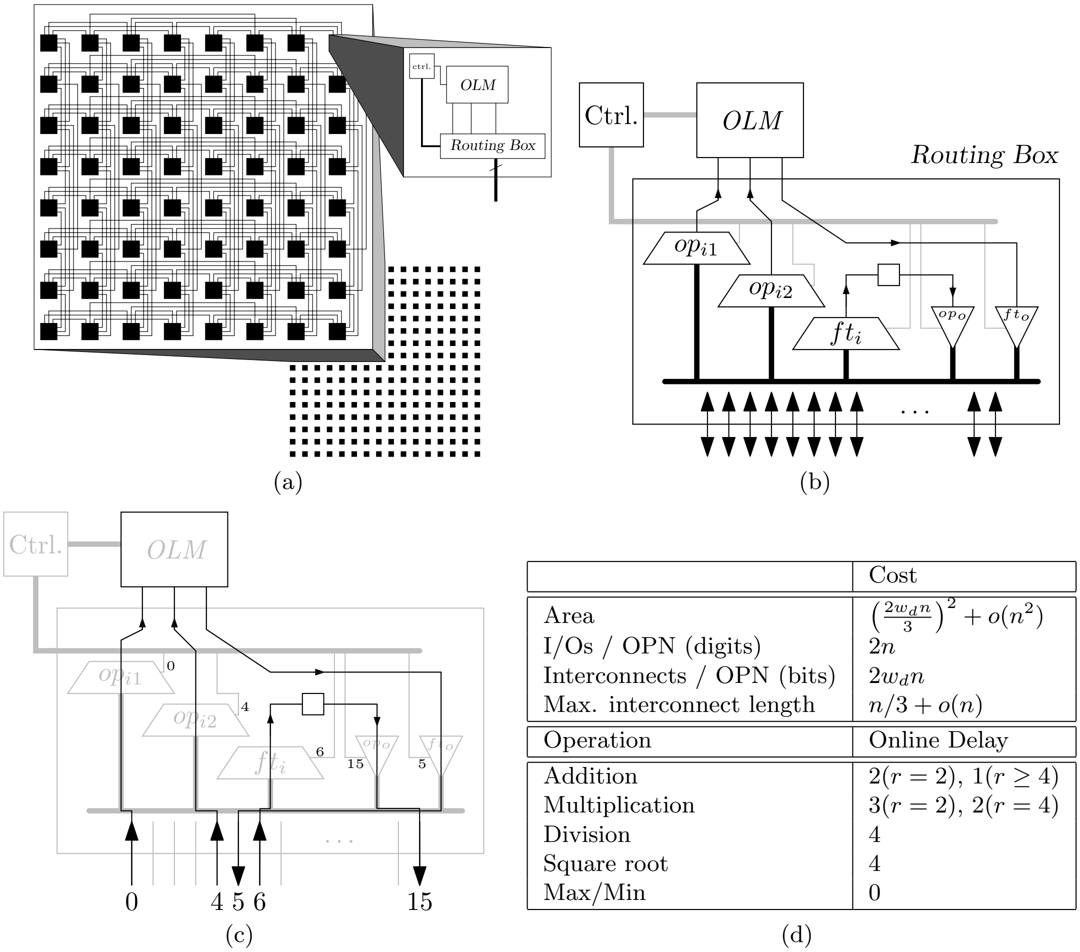
A general approach to compute arbitrary expressions interchangeably with little overhead is presented. Our approach relates to the paradigm of parallel computing platforms which *simulate* problems with given computational topologies within a general interconnection network of processing elements. We propose a hypercube of online processing nodes (OPN), on which we simulate any computational graph with a binary tree topology. We examine different tradeoffs in our proposed architecture such as the choice of radix and number of execution units.

3. ARCHITECTURE

A degree n hypercube network H_n consists of 2^n nodes labeled $n_0, n_1, \ldots, n_{2^n-1}$. A node n_i whose index i has binary representation i_2 has edges to all nodes n_j if the hamming distance (h) between i_2 and j_2 is one - that is $h(i_2, j_2) = h(n_i, n_j) = 1$. We define a *doubly-linked* hypercube H_n^2 as a hypercube which has two edges between any nodes n_i and n_j when $h(n_i, n_j) = 1$. The interconnection network of choice for OPNs is the doubly-linked hypercube.

If the set of expressions S_E contains an operation mix S_O, then each OPN is capable of performing all operations in S_o. An OPN has $2n$ I/O lines to neighboring nodes that carry a single digit at a time. Each digit in a radix-r redundant number system has an associated encoding which requires w_d bits–in radix 2 online arithmetic using a Borrow Save Digit (BSD) encoding ($1 \rightarrow 01, -1 \rightarrow 10, 0 \rightarrow 00$ or 11), two bits are required to encode digits ($w_d = 2$). This implies that an OPN in H_n^2 using radix-2 online arithmetic with BSD encoding requires $4n$ interconnects.

An OPN can perform a binary operation or a "feed-through"–where a single digit stream is routed through the OPN to another node. A routing box in the OPN selects two I/O digit streams as operands (op_{i1}, op_{i2}), and a single I/O onto which the result digit stream is output (op_o) with an optional feed-through from ft_i to ft_o as shown in Fig. 3(b). I/O paths must be chosen in the routing box such that digit streams are routed appropriately.



(a)

(b)

(c)

	Cost
Area	$\left(\frac{2w_d n}{3}\right)^2 + o(n^2)$
I/Os / OPN (digits)	$2n$
Interconnects / OPN (bits)	$2w_d n$
Max. interconnect length	$n/3 + o(n)$
Operation	Online Delay
Addition	$2(r = 2)$, $1(r \geq 4)$
Multiplication	$3(r = 2)$, $2(r = 4)$
Division	4
Square root	4
Max/Min	0

(d)

Figure 3. (a) Example architecture with $n = 8$ consisting of 256 OPNs. Wires in the hypercube layout represent two digit I/O paths. For a radix $r = 2$ implementation with Borrow Save Digit (BSD) encoding, each edge represents four physical wires. (b) Each OPN node has $2n$ digit connections, shown as bidirectional links to the routing box. A maximum of five links will be active at any point in time. (c) An instance of the routing box for H_8^2 where the OPN has 16 digit links. The inputs of the OPN are chosen via $op_{i1} = 0$ and $op_{i2} = 4$ and the output is driven on $op_o = 5$. This particular node also routes link 6 to link 15 as shown with setting $ft_i = 6$ and $ft_o = 15$.

The cost of implementation scales with the number of OPNs in the interconnection network in addition to the choice of radix which is used to perform serial arithmetic. The architecture's parameterized costs are shown in table 3(d). Layout costs are derived from [4] in which an efficient layout of the hypercube is presented.

The performance of evaluation in terms of cycles is dependent on the number of levels present in the network of online modules. Depending on the operation being performed at each node, a given online delay δ is in effect–examples of some online delays are shown in Fig. 3(d). Let Δ denote the *network online delay* as a single entity. Given all paths $r = p_0, \ldots, p_j = q$, where r is the root of the computational graph (CG), and q is a leaf node, the network online delay is given by (8). We define the total number of cycles required to compute CG as the *network delay* D.

$$\Delta = \max_{q \in CG} \left(\sum_{i=0}^{j} \delta(p_i) \right) \tag{8}$$

$$D = \Delta + m \tag{9}$$

The network delay is dependent on the type of expression being solved, for example, the fastest network is a min/max network since each node has an online delay of zero $\Rightarrow \Delta = 0$. Note that D does not take into account additional delays introduced by the mapping algorithm. The network delay is redefined in section to account for overhead delays associated with mapping characteristics.

4. MAPPING ALGORITHM

We review an algorithm here presented by Wagner[1] which embeds all binary trees in the Hypercube. This is performed in a two step process: **(1)** converting any arbitrary rooted binary tree with an even number of nodes to a strongly balanced tree *sbT*, and **(2)** finding a spanning of *sbT* in the hypercube. It is conjectured, that strongly balanced trees span the hypercube – no proof was presented but no counter examples were found. Part two of the algorithm labels vertices of the *sbT* obtained from part one to nodes in the hypercube. Characterization of the mapping is therefore entirely dependent on part one of the algorithm which is reviewed in Section 4.1. The reader is referred to [1] for part two of the algorithm.

The *guest* graph, an arbitrary rooted binary tree, will be embedded into the *host* graph, the hypercube. Given a guest graph $A = (V_A, E_A)$ and a host graph $B = (V_B, E_B)$, an embedding is defined as the mapping defined by $\phi : V_A \rightarrow V_B$. Characteristics of the mapping algorithm which are relied upon heavily are the maximum dilation, congestion and load of the embedding which are defined as follows. **(Dilation)** Given an edge $(v_{A1}, v_{A2}) \in E_A$ where $v_{A1}, v_{A2} \in V_A$, the *dilation* of an embedding ϕ is defined as the distance between the mapped nodes in the host graph, which is given by $h(\phi(v_{A1}), \phi(v_{A2}))$.**(Load)** The *load* of an embedding is the maximum number of nodes in the guest graph which map to a node in the host graph.**(Congestion)** The *congestion* of an edge is the number of paths in the guest graph which route through the edge in question within the host graph.

Converting an arbitrary binary tree to a strongly balanced tree causes some edges in the tree to be dilated. The maximum dilation of this embedding is two, hence no edge in the original binary tree will span more than two edges in *sbT*. If the routing algorithm in the hypercube can be chosen at runtime, each edge in the hypercube would have congestion at most two – two data paths occur on the same edge. These characteristics allow us to use the doubly-connected hypercube interconnection network to embed any expression. A doubly connected hypercube of degree three H_3^2 is shown in Fig. 4(a).

4.1. Strongly Balanced Trees

The algorithm reviewed requires that the binary tree T be rooted and have an even number of nodes – other cases are covered by adding a temporary node which is removed after the mapping. A strongly balanced tree *sbT* is a tree which contains a *perfect matching.* A matching M is a set of edges in T such that every node in the tree is the endpoint of exactly one edge in the matching. Consider the following two examples, shown in Fig. 4(d) and Fig. 4(e).

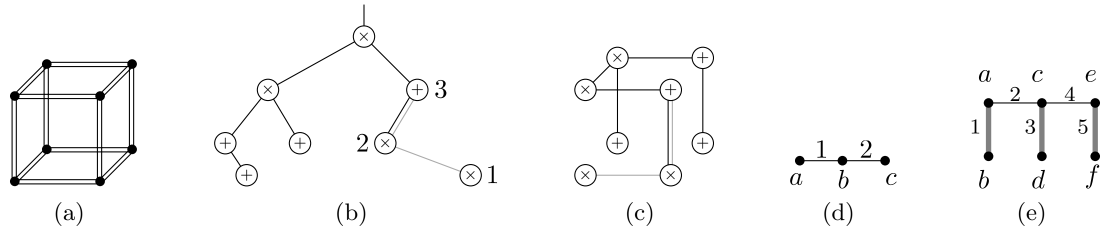


Figure 4. (a) A doubly-connected degree three hypercube: H_3^2 (b) Strongly balanced tree of the computational graph shown in Fig. 2(a). The light edge signifies that the result of operation 1 routes through node 2 to reach operation 3. (c) A spanning of the strongly balanced tree in H_3^2. (d) No perfect matching exists. Node b is shared by edges 1 and 2. (e) Perfect matching shown with thicker gray edges.

Edges in T are labeled *even* or *odd* to signify that removing the edge results in two components with an even or odd number of nodes respectively. Since T has an even number of nodes, several relationships can be deduced from the parity of edges.

Matching = {odd edges}: Let T be a tree with an even number of vertices. If T contains a perfect matching M then M equals the set of odd edges in the tree.

Equivalence relationships: Given T is a tree with an even number of nodes the following properties are equivalent: **(1)** T is strongly balanced. **(2)** Every vertex is incident to exactly one odd edge. **(3)** The number of odd edges is one more than the number of even edges. **(4)** Removing an even edge disconnects T into two strongly balanced trees.

4.2. Converting trees to strongly balanced trees

Two types of nodes can be identified in the original binary tree T: *3-0* nodes, which refer to nodes that contain 3 odd edges, and no even edges, and *1-x* nodes which refer to any node with a single odd edge, and $x \in \{0, 1, 2\}$ even edges. It is evident that both of these types of nodes preserve the condition that T has an even number of nodes. Also notice that no such thing as a 2-x node (two odd edges and x even edges) can exist, as it would contradict the even node count of T.

Since every vertex of an *sbT* is incident to exactly one odd edge, then converting all 3-0 nodes in T to 1-x nodes would yield *sbT*. All 3-0 nodes must be converted to 1-x nodes, and all 1-x nodes must be preserved. Consider a path $u \to v$ from u to v in T: $u = u_0, u_1, u_2, \ldots, u_k = v$, such that, **(1)** u is an ancestor of v **(2)** u is a 3-0 node and v is a leaf or a node of degree two **(3)** the path does not contain two consecutive even edges.

SHIFT$(u \to v)$ performs shifts along the path $\{u = u_0, u_1, \ldots, u_k = v\}$ such that the resulting path contains no 3-0 nodes. The shift operation is shown in Fig. 5(b). The shift operation dilates edges and introduces congested edges: After shifting, edge $\{s, u_i\}$ is dilated and comprises of two edges $\{s, u_i\}, \{u_i, u_{i+1}\}$ and edge $\{u_i, u_{i+1}\}$ has congestion 2.

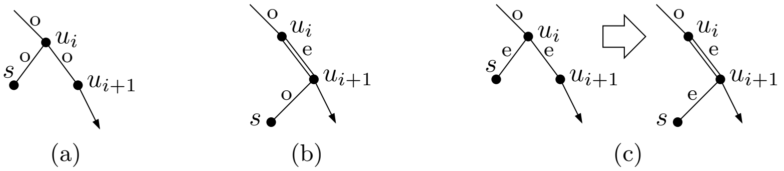


Figure 5. Given Path $u \to v$ containing nodes u_i and u_{i+1} (a) u_i: a 3-0 node. (b) Resulting u_i after shift is a 1-x node. (c) Performing a shift on node u_i where u_i is a 1-x node preserves u_i as a 1-x node. Note that only 1-2 nodes will be transformed in a shift; if node u_i was a 1-1 node, then u_{i+1} would have no sibling.

```
Algorithm 4.1: SHIFT($u \rightarrow v$)

comment: $sibling(x)$ returns the sibling of node $x$ or NIL
for $i \leftarrow 0$ to $k-1$
  do
    $sib \leftarrow sibling(u_{i+1})$
    if $sib \neq NIL$
      do
        remove edge $\{u_i, sib\}$
        add edge $\{u_{i+1}, sib\}$
```

The SHIFT() algorithm works by transforming all nodes on a path; 3-0 nodes are converted to 1-x nodes and 1-x nodes are preserved. If edge $\{u_i, s\}$ is odd and node u_i is a 3-0 node, then shifting node s to the child u_{i+1} changes the parity of edge $\{u_i, u_{i+1}\}$ to even as both components rooted at u_{i+1} and s are odd, as shown in Fig. 5(a) and 5(b). Preservation of 1-x nodes is shown in Fig. 5 The tree T is converted to sbT by performing SHIFT($u \rightarrow v$) on all paths $u \rightarrow v$ in T such that u is a 3-0 node and v is a leaf or vertex with degree two. The SHIFTPATHS() traverses T recursively, performing SHIFT() on all such paths. SHIFTPATHS() takes as parameters: r, the root of tree T, tree T, and a copy of T on which all transformations take place. When SHIFTPATHS() returns, the copy of T will be sbT.

```
Algorithm 4.2: SHIFTPATHS($r, T, sbT$)

procedure GETPATH(vertex $u$)
  $path \leftarrow \{\}, done \leftarrow False$
  while not $done$
    do
      $path \leftarrow \{path, u\}$
      if $u$ is a leaf
        then {return path
      $c_1, c_2 \leftarrow children(u)$
      if $\{u, c_1\}$ is odd
        then {$nextNode \leftarrow c_1$
        else if $\{u, c_2\}$ is even
        then {$nextNode \leftarrow c_2$
        else {$nextNode \leftarrow$ chose $nextNode$ randomly from $\{c_1, c_2\}$
      $u \leftarrow nextNode$

main
  if $r$ is a leaf
    then $return$
    else
      $path \leftarrow$ GETPATH($r$)
      $u \leftarrow$ first 3-0 node in $path$, else $NIL$
      if $u \neq NIL$
        then SHIFT($path$) on graph $sbT$
      for all $v$ in $V = \{$ siblings of all nodes on $path$ $\}$
        do SHIFTPATHS($v, T, sbT$)
```

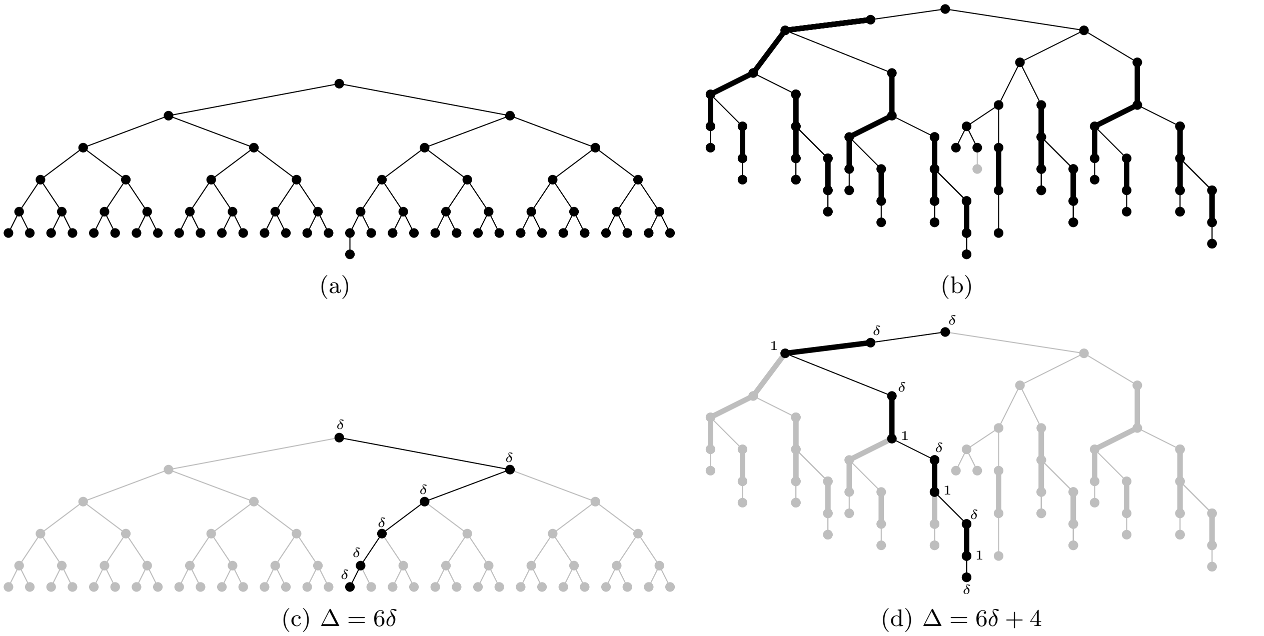
4.3. Network Delay

The network delay D presented in Section 3 did not take into account dilated edges which route through a sibling to the predecessor. These routes have an additional cycle of delay, which changes the network delay of the sbT resulting from T. Consider the dilated edge $\{s, u_i\}$ shown in Fig. 5(b): if u_i was the root of the expression, then the original delay (see Fig. 5(a)) from sibling s to u_i is given by $\delta(s) + \delta(u_i)$. However, once the edge is dilated, the new delay is given by $\delta(s) + 1 + \delta(u_i)$. In a similar manner through which the network online delay was presented in Section 3, an approach to finding the network online delay, and network delay of *sbT* is presented here.

Consider a path from the root of *sbT* to a leaf given by $r = p_0, p_1, p_2, ...p_{k-1} = q$. The network online delay of this path is given by eq. (11), and thus the network delay of the *sbT* resulting from *T* is given by eq. (12).

$$
\begin{aligned}
\delta d(p_i) &= \begin{cases} 1 & \text{, if } p_i \text{ routes } p_{i+1} \rightarrow p_{i-1} \text{ on the path} \\ \delta(p_i) & \text{, otherwise} \end{cases} && (10) \\
\Delta &= \delta(p_0) + \max_{q \in CG} \left(\sum_{i=1}^{k-2} \delta d(p_i) + \delta(q) \right) && (11) \\
D &= \Delta + m && (12)
\end{aligned}
$$

An example of a 6 level full binary tree *T* (with an added node to make its node count even) and its associated *sbT* are shown in Fig. 6(a) and Fig. 6(b). Assuming that each node has the same online delay δ, the paths from which the network online delays of *T* and *sbT* are obtained is shown in Fig. 6(c) and Fig. 6(d) respectively. Note that the extra node added to *T* to obtain an even node count is removed in *sbT* (the node must be disregarded *after* finding a labeling of *sbT* in H_n).



(a)

(b)

(c) $\Delta = 6\delta$

(d) $\Delta = 6\delta + 4$

Figure 6. (a) A full 6 level tree with an added extra node to satisfy the even node count requirement. (b) The corresponding *sbT* after ShiftPaths()is called on the root of the tree. (c) a path with largest online delay in the original tree (d) a path with the largest online delay in *sbT*. There is a 4 cycle overhead in the network online delay because of dilated edges introduced in the mapping.

5. DISCUSSION

In this section we discuss the key features and tradeoffs of our approach. We present a comparison between our approach to a commonly used approach utilizing h-trees.

H-trees are a layout strategy in which a full binary tree is realized through a series of "H" structures as shown in Fig. 7(c). An h-tree could be realized on an array of units as shown in Fig. 7(b). In a 16×16 array of units, an h-tree with a maximum of 7 levels can be realized, resulting in a maximum node utilization of 127/256. Note that this node utilization is the *maximum* and only occurs when a full binary tree with 7 levels is simulated on the interconnection of nodes. If an array is not required, then almost the same number of nodes (one less) can be used to construct an h-tree with an extra level as shown in Fig. 7(c). This gains an extra level in addition to making the maximum possible node utilization 255/255.

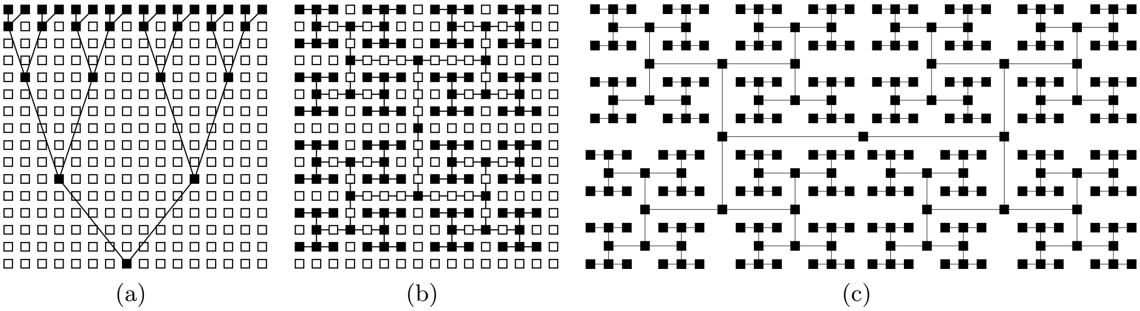


Figure 7. (a) An array of online arithmetic units with a simplistic approach to simulating a tree[5]; maximum node utilization: 31/256 (b) An array of online arithmetic units simulating an h-tree; maximum node utilization: 127/256. (c) An h-tree of online arithmetic units; maximum node utilization: 255/255

In the h-tree approach a simple observation is used to realize all interconnections of trees: all binary trees with maximum depth d_b span the full binary tree with depth d_f when $d_b \leq d_f$. That is, given a full binary tree with depth d_f, all binary trees with depth smaller than or equal to d_f are contained in the full binary tree. For example, consider the tree shown in Fig. 8(a) and its spanning in the h-tree.

The fundamental problem with the h-tree approach is that only trees with depth less than or equal to the h-tree span the interconnection of nodes. This *depth constraint* can cause severe underutilization of execution units in addition to breaking the chain of online operations. Consider the tree and its attempted spanning in the h-tree as shown in Fig. 8(b). Even though the tree requires much fewer execution units than available, the depth of the tree can not be accommodated by the h-tree forcing the expression to be computed as two separate online chains–the black dots on the right hand side denote the nodes which could not be mapped into the h-tree. When computing the expression as two online chains, the overall delay of computing the expression is given by $\Delta_1 + \Delta_2 + 2m$ where Δ_1 and Δ_2 are the network online delays of the first and second online chains respectively. However, computing the expression as a single online chain has delay $\Delta_1 + \Delta_2 + m$; therefore, it is critical to avoid breaking the online chain. However, the depth constraint of h-trees forces breaking the online chain every time the depth of the h-tree is exceeded.

The depth constraint of h-trees in the spanning of subtrees is an inappropriate constraint as the fundamental constraint of computability should be the availability of execution units. In our proposed approach the only

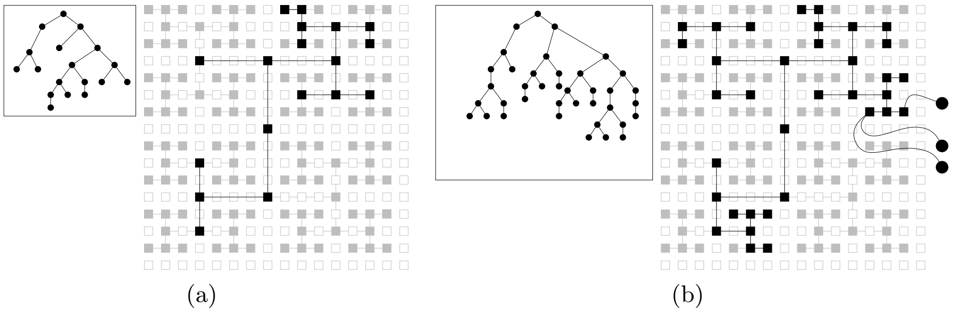


Figure 8. (a) Mapping of shown CG in the h-tree. (b) Attempted mapping of shown CG in the h-tree–unsuccessful due to depth constraint.

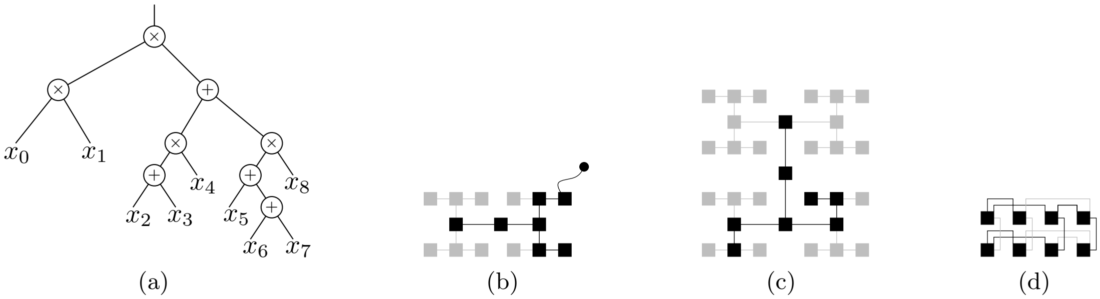


Figure 9. (a) The computational graph (b) Attempted mapping to h-tree (c) Successful mapping to h-tree (d) Mapping to proposed interconnection network.

constraint in simulating trees within the doubly-linked hypercube network is the number of available execution units–our algorithm is *resource constrained.* This means that we will always achieve 100% utilization of resources if an expression is large enough to require all OPN nodes. For example, consider an expression requiring 235 operations with any arbitrary tree structure. Our approach can embed this expression into H_8^2 leaving 21 nodes unused as the expression does not require all 256 resources. If the expression required all 256 nodes, then all 256 nodes could be allocated. If the expression required more nodes than available, then the expression would have to be partitioned into smaller subexpressions with resource requirements less than or equal to the available number of resources.

For example, consider the expression $E = x_0x_1((x_2+x_3)x_4+(x_5+x_6+x_7)x_8)$ whose CG is shown in Fig. 9(a). Mappings to the h-tree and our proposed interconnection of OPNs is shown in Fig. 9(b), Fig. 9(c) and Fig. 9(d).

Another important property of these interconnection schemes is their ability to *pack* smaller trees. Consider a scenario where several smaller expressions are to be evaluated; if these expressions can be "packed" into the interconnection of online arithmetic units then they could be evaluated simultaneously.

The h-tree was not very suitable for a single expression to begin with, which means that it would be at best as inefficient or worse at packing smaller trees. When using an h-tree, the full binary tree can be viewed as a number of smaller full binary trees on which the expressions can be evaluated. The inefficiency of depth restricted spanning is once again seen on the smaller sub-h-trees of the interconnection network. The inefficiency of spannings in sub-h-trees is aggregated to yield an inefficient packing.

In our approach, a larger tree could be constructed from the smaller trees of expressions to be evaluated as shown in Fig. 10. The resulting tree is mapped to the interconnection of OPN nodes and a labeling of all vertices is obtained. The large tree can then be dissected back to the original expressions to be evaluated. Notice that after adding an edge between a leaf and the root of another tree (e.g. E_i and E_t) the leaf node has no siblings. This means that SHIFT() does not affect the leaf node, leaving the two trees distinct after mapping. Once the large tree is mapped, each original smaller expression's result is obtained at the root of the appropriate subtree.

It is difficult to contrast the efficiency of h-trees and our approach in general, however, a fair but skeptical analysis would be to analyze how much worse our approach performs what the h-tree does best. The h-tree has the highest resource utilization for a full binary tree with the same depth as the h-tree–which we denote as l. If the online delay of all nodes is the same (δ), then the h-tree can perform the associated computation in $l\delta + m$ cycles, where m is the precision of the computation. In our approach, since there are l levels and $l - 1$ edges from the root to a leaf, then $l\delta + q + m$ cycles are required, where $q \leq l - 1$ and depends on the number of dilated edges on the path. This additional delay is not drastic considering that l is logarithmic with respect to the number of operations performed.

The tradeoff in evaluating expressions using the proposed interconnection scheme is the additional cycles of network online delay introduced by dilated edges in *sbT*. The cost of implementation for the proposed interconnection scheme is also higher than an h-tree with an equivalent number of execution units. This cost is attributed to the additional interconnects required by the doubly linked hypercube.

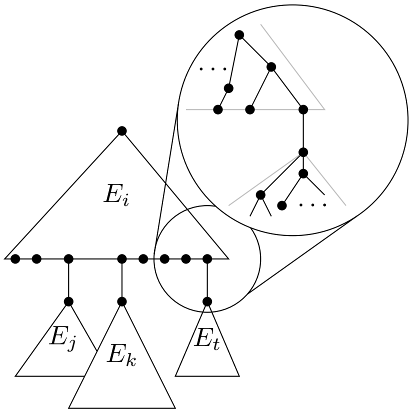


Figure 10. Forming a larger tree from trees of smaller expressions creates an efficient packing of the smaller expressions. After mapping of the large expression, each subexpression's result is obtained from the node with label corresponding to the root of the subexpression.

6. SUMMARY

In this paper we proposed an interconnection scheme for online arithmetic operators which facilitates evaluation of all arithmetic expressions whose computational graph has a tree topology. Our approach achieves high utilization of execution units to exploit the most parallelism from a given expression with little overhead. For future work we aim to create a hardware implementation of proposed architecture through which detailed resource/delay costs can be extracted, and contrasted to existing approaches. We also plan to investigate the interplay between the choice of radix and the associated hardware cost/delay.

REFERENCES

1. Alan S. Wagner, "Embedding All Binary Trees in the Hypercube", *Journal of Parallel and Distributed Computing*, Vol. 18, pp. 33–43, 1993.
2. M.D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann Publishers, San Francisco, 2004.
3. 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.
4. C. Yeh, E. A. Varvarigos, B. Parhami, "Efficient VLSI Layouts of Hypercubic Networks",Proceedings of Frontiers '99: The 7th Symposium on the Frontiers of Massively Parallel Computation, pp. 98-105, February 21-25, 1999.
5. M. F. Aguilar, "Conception et Simulation d'une Machine Massivement Parallèle en Grande Précision", PhD thesis, L'Ecole Normale Supérieure de Lyon, France, 1994.