Area-Efficient Arithmetic Expression Evaluation Using Deeply Pipelined Floating-Point Cores

Ronald Scrofano, *Member, IEEE*, Ling Zhuo, *Member, IEEE*, and Viktor K. Prasanna, *Fellow, IEEE*

***Abstract*—Recently, it has become possible to implement floating-point cores on field-programmable gate arrays (FPGAs) to provide acceleration for the myriad applications that require high-performance floating-point arithmetic. To achieve high clock rates, floating-point cores for FPGAs must be deeply pipelined. This deep pipelining makes it difficult to reuse the same floating-point core for a series of dependent computations. However, floating-point cores use a great deal of area, so it is important to use as few of them in an architecture as possible. In this paper, we describe area-efficient architectures and algorithms for arithmetic expression evaluation. Such expression evaluation is necessary in applications from a wide variety of fields, including scientific computing and cognition. The proposed designs effectively hide the pipeline latency of the floating-point cores and use at most two floating-point cores for each type of operator in the expression. While best-suited for particular classes of expressions, the proposed designs can evaluate general expressions as well. Additionally, multiple expressions can be evaluated without reconfiguration. Experimental results show that the areas of our designs increase linearly with the number of types of operations in the expression and that our designs occupy less area and achieve higher throughput than designs generated by a commercial hardware compiler.**

***Index Terms*—Expression evaluation, pipeline arithmetic.**

I. Introduction

MANY computationally intensive applications that are good candidates for hardware acceleration require high precision, floating-point arithmetic. Recent advances in field-programmable gate array (FPGA) technology have made FPGAs a potential alternative for accelerating floating-point applications. The emergence of reconfigurable computers that have both general purpose processors and FPGAs, as well as high performance interconnect between them, has also contributed to making FPGAs an attractive platform for the acceleration of floating-point applications.

The arithmetic expression evaluation problem has long been of theoretical and practical interest [1]. It is the problem of computing the value of an expression that is represented by a tree in which the leaves are numeric values and the internal nodes are operators. Many efficient algorithms have been proposed in the parallel computing community [2], [3]. With the increasing computing power of FPGAs, multiple floating-point cores can be configured on one FPGA and work cooperatively. Thus, an interesting problem is the development of FPGA-based designs that provide acceleration for arithmetic expression evaluation.

However, developing high-performance floating-point designs on FPGAs is still challenging. Because of their complexity, the floating-point cores have very large areas. Therefore, it is impractical for a design to use $\Theta(n)$ floating-point cores, where n is the number of numerical values in an expression. Instead, it is desirable to use as few of the cores in a design as possible. However, such area efficiency requires reuse of the same floating-point core for a series of computations that are dependent upon one another. As the floating-point cores are deeply pipelined to achieve a high clock frequency, such reuse may cause read-after-write data hazards. On the other hand, if the pipeline of a floating-point design is stalled to avoid data hazards, its throughput will be adversely affected. Therefore, a design problem is to develop algorithms and architectures that can hide the pipeline latency and use as few floating-point cores as possible in a design while still maintaining a high throughput.

In this paper, we propose a high-performance and area-efficient solution to the arithmetic expression evaluation problem. In our proposed architectures, the number of floating-point cores needed is independent of n and the buffer size is $\Theta(\lg(n))$. The first proposed design is most efficient for expressions whose expression trees are complete binary trees: trees in which all the internal nodes have degree 2 and all the leaves have the same depth. We show that if the n inputs to such an expression arrive sequentially, it is possible to evaluate the expression with only one floating-point core of each type used in the expression and complete the evaluation in $\Theta(n)$ time. For expressions whose expression trees are not complete binary trees but satisfy certain requirements, we propose a similar design that uses two floating-point cores of each type. We then discuss the penalty for general expressions that cannot be directly evaluated by the proposed designs. We also describe how the design can evaluate multiple expressions without the need for hardware reconfiguration.

We have implemented the proposed solution on Xilinx Virtex-II and Virtex-II Pro FPGAs. The area of our architecture increases linearly with the number of types of operators. On the other hand, when the number of operators is fixed, there is only a small area increase as the number of inputs increases. We compare the proposed solution with a straightforward hardware design and compiler-generated designs.

The rest of this paper is organized as follows. In Section II, we define and illustrate the arithmetic expression evaluation problem and the design challenges that it presents. In

Manuscript received May 20, 2006; revised February 26, 2007. This work was supported in part by Los Alamos National Laboratory under Contract/Award 95976-001-04 3C and by the National Science Foundation (NSF) under Awards CCR-0311823 and ACI-0305763. Some development software and equipment was purchased with NSF equipment Grant CNS-0454407.

R. Scrofano was with the Computer Science Department, University of Southern California, Los Angeles, CA 90089-0781. He is now with the Computer Systems Research Department, The Aerospace Corporation, Segundo, CA 90245-4691 USA.

L. Zhuo and V. K. Prasanna are with the Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089-2562 USA (e-mail: rscrofan@halcyon.usc.edu).

Digital Object Identifier 10.1109/TVLSI.2007.912038

1063-8210/$25.00 © 2007 IEEE

Section III, we describe the proposed algorithms and architectures for arithmetic expression evaluation and we prove their correctness. Section IV presents analysis of our design's performance. In Section V, we discuss related work. Finally, Section VI concludes the work and presents areas for future study.

II. Arithmetic Expression Evaluation

A. Problem Definition

An arithmetic expression is any well-formed string composed of at least one of the arithmetic operations, left and right parentheses as needed and numerical values that are either constants or variables. We denote an arithmetic expression e of n distinct numerical values by $e(n)$. Suppose these n values are input sequentially as $[r_0, r_1, \ldots, r_{n-1}]$; and one r_i comes in per clock cycle $(0 \leq i \leq n-1)$. The arithmetic expression evaluation problem is thus to compute the value of e applied to these n inputs. Also, we use S to denote the set of types of operations in the expression.

B. Tree Representation

An arithmetic expression can be represented by a tree in which the leaves are numeric values and the internal nodes are operators. In particular, most expression trees can be transformed into a binary tree. Unary operators can be considered with an identity operation. [4], on the other hand, describes ways to transform into binary trees those expression trees in which there are nodes for associative, commutative operators that take more than two inputs.

There also have been some works on arithmetic expression tree-height reduction. In [5], an upper bound on the reduced tree height is given assuming that only associativity and commutativity are used. Let $e(n)$ be any arithmetic expression with depth d of parenthesis nesting. The tree of $e(n)$ can be transformed in such a way that its height is less than or equal to $\lceil \lg n \rceil + 2d + 1$. Bounds using associativity, commutativity and distributivity have also been given. In [2], it is proven that the tree height of $e(n)$ can be reduced to under $\lceil 4 \lg n \rceil$. For expressions of special forms, better bounds can be given, as shown in [6].

Therefore, in our paper, we focus on expressions that can be represented by binary trees. Additionally, the trees are balanced, that is, the heights of the subtrees of any node differ by only a small amount. "Skinny" trees or nonbinary trees can be transformed using existing methods.

C. Applications

In many scientific computing algorithms, the inner loops need to evaluate arithmetic expressions. Molecular dynamics—a technique for simulating the trajectory of a system of atoms by integrating the classical equations of motion—has several arithmetic expressions that must be evaluated many times. Such expressions usually have tens of inputs. Fig. 1 shows an example expression from the force calculation phase. All of the leaves in the expression tree are either constants or are calculated prior to this expression's evaluation.

The proposed design is applicable to other scientific kernels, as well, including vector dot product and matrix-vector multiplication. These kernels require the accumulation of floating-point numbers. Such accumulation is a special case of arithmetic expression evaluation. For large-scale problems, these kernels have expressions with thousands of inputs.

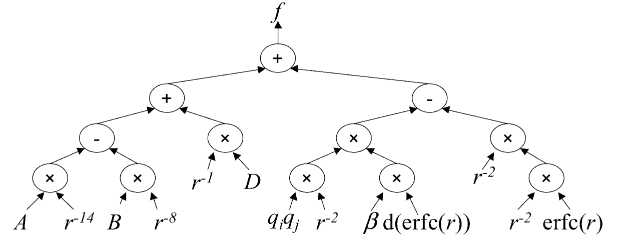

Fig. 1. Example expression tree for molecular dynamics force calculation.

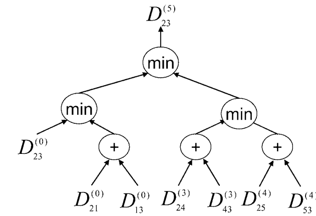

Fig. 2. Example expression tree for the Floyd–Warshall algorithm.

Probabilistic inference also uses expression evaluations. For example, in the Lauritzen-Spiegelhalter algorithm, a general belief network is transformed into a tree of cliques [7]. To find the probability of events, expressions must be evaluated at each node and the results propagated up the tree.

Besides floating-point scientific applications, arithmetic expression evaluation exists in many other applications. For example, the graph theoretical problems of all-pairs shortest-path and transitive closure are fundamental in a variety of fields. These problems are solved by the Floyd–Warshall algorithm, which requires comparisons and additions, either fixed- or floating-point, depending upon the problem domain. The Floyd–Warshall algorithm for a graph of n vertices consists of n steps. We use $D_{ij}^{(k)}$ to denote the weight of the shortest path from vertex i to vertex j in step $k (i, j, k = 1, \ldots, n)$. $D^{(0)}$ is the weight matrix of the original graph. In the algorithm, $D_{ij}^{(n)} = \min(D_{ij}^{(0)}, D_{i1}^{(0)} + D_{1j}^{(0)}, D_{i2}^{(1)} + D_{2j}^{(1)}, \ldots, D_{i,n}^{(n-1)} + D_{n,j}^{(n-1)})$. Suppose the $\min$ function is implemented using a binary comparator. Fig. 2 shows the expression tree for $D_{23}^{(5)}$ when $n = 5$.

D. FPGA-Based Arithmetic Expression Evaluation

We now discuss the design issues for arithmetic expression evaluation on FPGAs. To begin with, we restrict the trees to complete binary trees with four or more inputs. Thus, a tree with k levels has $n = 2^k$ inputs and each of the operators is a binary operator. An example of such a tree is shown in Fig. 3.

The evaluation problem for such expressions is trivially solved with $n - 1$ floating-point cores, one for each operation in the expression. However, this is a very inefficient solution. The large areas of floating-point cores make such a solution undesirable or, for large n, infeasible. When the inputs to the expression arrive sequentially, such an architecture also makes uneconomical use of the floating-point cores. The floating-point cores in the architecture do not take new inputs on every clock cycle; rather, they are only active periodically. For example, the first floating-point core at level 0 of the tree operates on the first two inputs to arrive. Before it can operate on any more inputs,

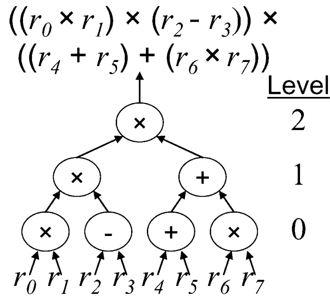

Fig. 3. Example of an expression tree for an expression with eight inputs.

it must wait the $n-2$ cycles it takes for the next $n-2$ inputs to arrive and enter the other floating-point cores in level 0, plus two more cycles for its new inputs to arrive. Such inefficiency exists at all levels of the tree.

An architecture like that of Fig. 4 is a first step in addressing this inefficiency. There is only one floating-point core of each type necessary in a given tree level, a constant-size buffer, and a multiplexer. We assume throughout this paper that the number of pipeline stages in each floating-point core is the same and we call it α (in Section III-C we describe what to do when this is not the case). There are now $O(|S|\lg(n))$ floating-point cores and $\lg(n)$ constant-size buffers. At each level of this compacted tree, when there are two values in the buffer, they are read from the buffer and passed to the floating-point cores. The correct floating-point core output, as chosen by the multiplexer's selection logic, is written to the buffer at the next highest level.

While an improvement over the trivial solution, this solution is still inefficient. The floating-point cores at each level are still not taking new inputs at each clock cycle. The floating-point cores at level 0 take new inputs every other clock cycle. The floating-point cores at level 1 must wait for two outputs of the floating-point cores at level 0. So, these floating-point cores only read new inputs once every four cycles. The floating-point cores at level i only read new inputs once every 2^{i+1} cycles. Additionally, this architecture requires $O(|S|\lg(n))$ floating-point cores, which still may not be feasible for large values of n. But, since the floating-point cores are not fully utilized, we see that with a clever architecture and algorithm, it is possible to evaluate the expression using a single floating-point core for each type of operator in the expression.

The difficulty in developing such an architecture and algorithm lies in the fact that floating-point cores on FPGAs are very deeply pipelined. Partial results are not available in the cycle immediately following the start of their computation. Instead, they are available α stages later. If $\Theta(n)$ storage is allowed, the inputs can be buffered until a partial result is ready. Once again, though, this solution leads to inefficient use of the floating-point cores. Additionally, stalling for each operation will drastically reduce throughput. Since we are targeting applications in which the operands will be double-precision floating-point numbers, we do not want to have to store many values on the chip. We anticipate that expression evaluation will be part of a larger application implemented in hardware. So, while there is sufficient memory on current FPGA devices to store a substantial number of double-precision floating-point numbers, we would like to use as little memory as possible for the expression evaluation and leave the rest free for the other parts of the application. Thus, we would like to use $O(\lg(n))$ buffer space.

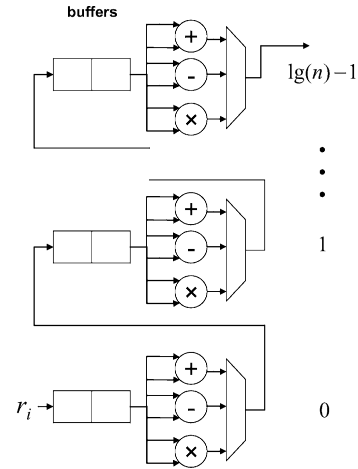

Fig. 4. Complete binary tree for an expression with three types of operators after compaction to one floating-point core of each type per level ($+$, $-$, and $\times$ could be replaced by any binary operators).

In summary, the focus of our work is to develop designs for arithmetic expression evaluation on FPGAs using the minimum number of deeply pipelined floating-point cores possible. Additionally, the proposed solution should not require $\Theta(n)$ buffer size or introduce pipeline stalls.

III. Proposed Algorithms and Architectures for the Arithmetic Expression Evaluation Problem

In this section, we present the proposed solution to the arithmetic expression evaluation problem. We start with the evaluation of expressions whose expression trees are complete binary trees, which is referred to as the "basic case." We then extend this solution to the "advanced case," where more general expressions are considered. For each case, we prove the correctness of the proposed solution, and note some important features of it. At the end of this section, we discuss how our proposed designs can be used for general expressions.

A. Basic Case

1) Architecture and Algorithm: As described previously, the floating-point cores at level i in the architecture in Fig. 4 are utilized only once every 2^{i+1} clock cycles. With careful scheduling, then, we can interleave all floating-point operations in a single set of $|S|$ floating-point cores. A selection function is used to decide which floating-point core's output is written to buffer $i+1$. In that way, it is possible to select which floating-point core's result should be used in the later calculations. By selecting the appropriate result for each buffer write, any complete-binary-tree expression e can be evaluated. We formalize these ideas in the following.

The architecture for the solution to the basic case is shown in Fig. 5. It has one α-stage pipelined floating-point core for each type of operation in the expression, a counter (C), some control circuitry, and $\lg(n)$ buffer levels. All of the buffers hold three data items, except for the buffer at level 0, which only holds two data items. Note that the buffers are first-input–first-outputs (FIFOs). The r_i values are written into buffer 0 at every clock cycle. When there are two values in the buffer, they are destructively read and are presented as inputs to the floating-point cores. Two cycles later, the next two r_i values are presented as inputs to the floating-point cores. When the floating-point cores

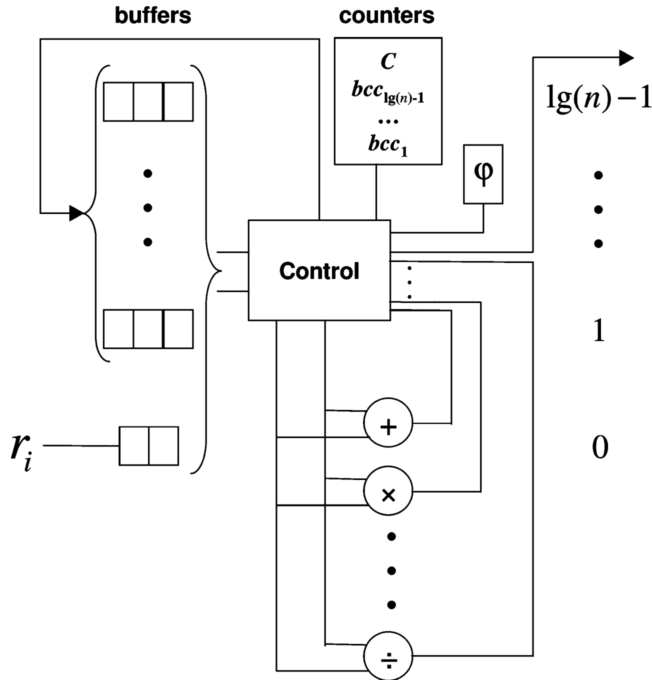

Fig. 5. Architecture for the basic case ($+, \times, \div$ could be replaced by any binary operators).

have operated on two values from buffer i, one floating-point core's output is chosen and is written to buffer $i+1$. To select the output from the correct floating-point core, the architecture has $\lg(n)-2$ *buffer control counters* $bcc_1, \ldots, bcc_{\lg(n)-1}$, and has a selection function φ.

Buffer $i, i = 1, \ldots, \lg(n)-1$, has a $(\lg(n)-i)$-bit buffer control counter, bcc_i, associated with it. The purpose of bcc_i is to allow buffer i to decide which floating-point-core output should be written to it. bcc_i counts the number of writes to buffer i that have occurred. For example, consider the tree in Fig. 3. There are three types of operators in level 0 of the tree: $\times, -$, and $+$. Each of these operators has a corresponding floating-point core in the architecture for this expression. When data is to be written into buffer 1, buffer 1 must be able to correctly choose which floating-point core's output should be written into it. From Fig. 3, one can clearly see that the first write to buffer 1 should be from the multiplier, the second write should be from the subtracter, and so on. Thus, when $bcc_1 = 0$, the output of the multiplier core should be written to buffer 1, when $bcc_1 = 1$, the output of the subtracter core should be written to buffer 1, and so on. This mapping between bcc_i and the functional units is handled by the selection function φ.

φ is the function that controls the selection of floating-point core outputs to be written into a given buffer. φ has two parameters: the buffer level and the value of the buffer control counter for that level. The output of φ is the type of operator whose output should be written into the buffer. For example, for the tree in Fig. 3, $\varphi(1,0) =$ *multiplier* and $\varphi(1,1) =$ *subtracter*.

The algorithm for the solution to the basic case is given in Fig. 6. It describes the schedule for reading from and writing to the buffers, the buffer control counters and the selection function. The sections of the algorithm are executed in parallel every clock cycle. The notation $v_{<j>}$ is used to denote the j least significant bits of v. For notational convenience, we define the value of $\varphi(\lg(n))$ to be the floating-point core for the type of operator that is at the root of the complete binary tree.

2) Proof of Correctness: In this section, we prove several lemmas and theorems to show that the proposed algorithm and architecture solve the basic case of the problem.

Lemma 1: $\Theta(\lg(n))$ buffer space is sufficient for the algorithm and architecture and there are no data hazards in the architecture.

Proof: We first prove that $\Theta(\lg(n))$ buffer space is sufficient for the proposed algorithm and architecture. For buffer i, one value is written every 2^i clock cycles; two values are read every 2^{i+1} clock cycles because the floating-point cores in the architecture are all binary operators. Thus, we have $\text{rate}_{\text{write}} = \text{rate}_{\text{read}}$, and a buffer cannot grow without bound once we begin reading values from that buffer.

Obviously, buffer 0 at most contains two values before it is first read because it is read every two clock cycles. We now prove that all the other buffers at most contain three values before they are first read. There are $2^{i+1}-1$ clock cycles between two reads of buffer i. When $i > 0$, we have $2^i+1 < 2^{i+1}-1 < 2 \times 2^i + 1$. Thus, during $2^{i+1}-1$ clock cycles, at most two values are written into buffer i. Suppose buffer i is first read at t_i. If buffer i contains more than three values, then it would have had two or more values at $t_i - 2^{i+1}$, which means that it should have been read earlier and t_i is not the first read. This contradicts our assumption. Thus, buffer i needs at most three values and $\Theta(\lg(n))$ buffer space is sufficient.

We now prove by contradiction that no data hazards occur in the proposed architecture. Assume a hazard occurs when the floating-point cores try to read from buffer i before the correct value is written to buffer i. If the read succeeds, we know that buffer i contains two valid values and this is not an invalid read. If the buffer contains less than two valid values, the read will not succeed and the hazard will not occur. In both cases, we reach the contradiction and the lemma is proved. □

Lemma 2: The partial results from different input sets are not combined together during computation.

Proof: Let R and T be two sets of n inputs. The elements of an input set arrive at the architecture sequentially. Without loss of generality, let the elements of R be the first to arrive. Since elements arrive sequentially, one per clock cycle, all of the elements of R arrive and enter buffer 0 before any of the elements of T do. The floating-point cores are linear pipelines of the same length, α. Since the elements of T enter the floating-point cores after the elements of R, they must exit the floating-point cores after the elements of R. The first two elements to enter buffer i are the first two elements to leave buffer i. Thus, for a partial result from R to be combined with a partial result from T, there must be exactly one partial result from R and one or more partial results from T in buffer i, for some i. However, the number of inputs in R is $n = 2^k$ for some integer k. So, at level i, there are $(n)/2^i = 2^{k-i}$ partial results. This is an even number. The algorithm calls for the floating-point cores to read two values from buffer i at a time. Thus, it is impossible for exactly one partial result from R to be left in the buffer with one or more partial results from T. Therefore, it is not possible for operands from multiple sets to be combined together during computation. □

Theorem 3: The algorithm and architecture together correctly evaluate an expression e whose expression tree is a complete binary tree with $n = 2^k$ inputs for all $k \geq 2$.

```
{counters}
if floating-point cores are first used then
  C = 0
  for i = 1 to lg(n) − 1 in parallel do
    bcc_i = 0
  end for
else
  C = C + 1
end if
{buffers}
write input port to buffer 0
for i = 0 to lg(n) − 2 in parallel do
  if (C − α)⟨i+1⟩ = 2^i − 1 then
    write output of floating-point core φ(i + 1, bcc_{i+1}) to
    buffer i + 1
    bcc_{i+1} = bcc_{i+1} + 1
  end if
end for
{floating-point cores}
for i = 0 to lg(n) − 1 in parallel do
  if C⟨i+1⟩ = 2^{i-1} and
  buffer i has more than 1 value then
    read 2 values from buffer i
    for all s ∈ S in parallel do
      Enter the two values into floating-point core s
    end for
  end if
end for
{output}
if output of floating-point cores is valid then
  if (C − α)⟨lg(n)⟩ = n/2 − 1 then
    write the output of floating-point core φ(lg(n)) to
    external memory
  end if
end if
```

Fig. 6. Algorithm for the basic case.

Proof: We prove this theorem inductively. First, we assume that e has $n = 2^2 = 4$ inputs and show that the algorithm works. Let R be the set of n inputs with r_0 the first to arrive, r_1 the second to arrive, and so on. When r_0 arrives, it is stored in buffer 0. When r_1 arrives, it is also stored in buffer 0. On the next cycle, C is set to 0 and r_0 and r_1 are given as input to all of the floating-point cores. Also, r_2 is written to buffer 0. On the next cycle, r_3 is written to buffer 0. On the next cycle after that (C is 2, so $C_{\langle 1\rangle} = 2^0 - 1$), r_2 and r_3 are read from the buffer and given as inputs to the floating-point cores. α stages after r_0 and r_1 entered the floating-point cores, $(C - \alpha)_{\langle 1\rangle} = 2^0 - 1$, so the output of one of the floating-point cores must be written. $bcc_1 = 0$, so the output of floating-point core $\varphi(1, 0)$ is written to buffer 1. By the definition of φ, this is the correct output to be written. bcc_1 is incremented to 1. Two cycles later, $(C - \alpha)_{\langle 1\rangle} = 2^0 - 1$ again, so it is time to write to buffer 1 again. Since r_2 and r_3 were given as inputs to the floating-point cores two cycles after r_0 and r_1 were, their partial result is the output from the floating-point cores. $bcc_1 = 1$, so the output of the floating-point core $\varphi(1, 1)$ is written into buffer 1. bcc_1 is incremented and rolls over back to 0. Buffer 1 now has two inputs, so as soon as $C_{\langle 2\rangle} = 01$, the two partial results will be read from buffer 1 and given as inputs to the floating-point cores. α cycles after that, the output is ready. Correspondingly, $(C - \alpha)_{\langle 2\rangle} = 4/2 - 1 = 01$, so the output of floating-point core $\varphi(\lg(n))$ is written to external memory. By definition, this is the output from the correct floating-point core. So, the output is correct.

For the inductive hypothesis, we assume that the algorithm and architecture produce the correct results for $n = 2^k$, $k > 2$, inputs. We now prove that, if that is the case, the algorithm and architecture produce the correct results for $n = 2^{k+1}$ inputs.

The key observation is that evaluating an expression of $n = 2^{k+1}$ inputs is almost the same as evaluating two consecutive expressions of $m = 2^k = n/2$ inputs. For n inputs, an extra level of buffering is added to the architecture and one more operation must take place. Also, φ and bcc_i are extended accordingly and bcc_{n-1} is added to the architecture. The first m inputs to arrive are operated on in the same manner as they would have been if the algorithm and architecture were designed for m inputs. The only difference is that instead of the output of floating-point core $\varphi(\lg(m))$ being written to external memory when $(C - \alpha)_{\langle \lg(m)\rangle} = m/2 - 1$, the output of what is now $\varphi(\lg(m), 0)$ is written to buffer $\lg(n) - 1$. By the inductive hypothesis, this value is the correct evaluation of the left subtree of the expression. Similarly, the right subtree of the expression is also correctly evaluated. We know from Lemma 2 that the partial results from two sets of m inputs could not have been combined together during the calculation. The values from the left and right subtrees have been written into buffer $\lg(n) - 1$. As soon as $C_{\langle \lg(n)\rangle} = 2^{\lg(n-1)} - 1 = n/2 - 1$, those two values are read from buffer $\lg(n) - 1$ and given as inputs to the floating-point cores. α cycles later, when $(C - \alpha)_{\langle \lg(n)\rangle} = n/2 - 1$, the output of $\varphi(\lg(n))$ is written to external memory. By definition, this is the output of the correct floating-point core. Therefore, the correct result is produced.

Thus, the algorithm and architecture together correctly evaluate an expression e of $n = 2^k$ inputs for all $k \geq 2$. □

Theorem 4: The latency of the algorithm is at most $3n + (\alpha - 1)\lg(n) - 2$ cycles.

Proof: It takes n cycles for all the inputs to arrive. The last input will go through the floating-point cores $\lg(n)$ times. It will spend at most $2^{i+1} - 1$ cycles in buffer i. So the total latency is at most $n + \alpha \lg(n) + \sum_{i=0}^{lg(n)-1}(2^{i+1} - 1) = 3n + (\alpha - 1)\lg(n) - 2$ cycles. □

B. Advanced Case

We now extend the architecture for the basic case to the advanced case. In this case, the number of inputs n does not need to be a power of 2, and the expression tree does not need to be a complete binary tree. However, we assume that at each expression tree level, all values must be operands of some binary operators except the last one. If the last value is not paired with any other value, it is called a "singleton value."

When n is not a power of 2, there may be operations on singleton values in multiple levels of the tree. The architecture in Fig. 5 is not able to handle the singleton operations for the following reason. When we pad a singleton operation with an identity, values in the subsequent sets are still arriving in the buffers. Since only the singleton value is consumed from the buffer instead of two values, the buffers will overflow.

1) Algorithm and Architecture: The architecture in Fig. 7 is proposed. This architecture contains two sets of floating-point cores and $\lceil \lg(n) \rceil$ buffers. By using an additional set of

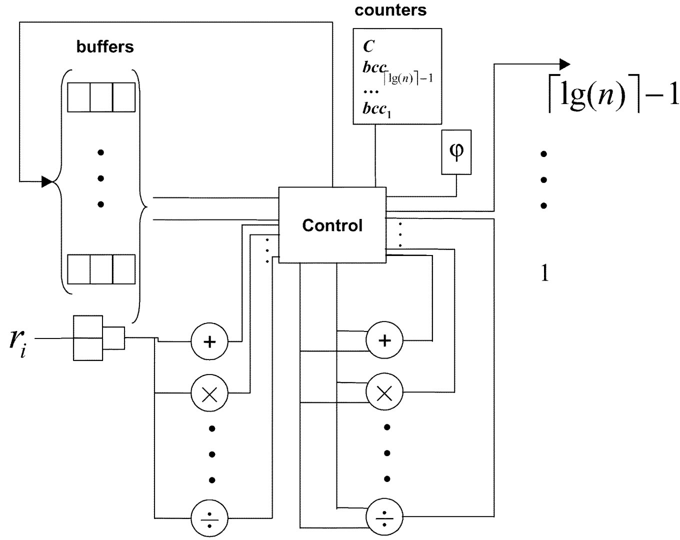

Fig. 7. Architecture for the advanced case ($+, \times, \div$ could be replaced by any binary operators).

floating-point cores, the architecture provides a sufficient number of available pipeline slots to schedule the "extra" operations precipitated by the singleton cases. In particular, the first set of floating-point cores handles the operations in the first level of the binary tree, while the second set of floating-point cores is in charge of the operations in all the other levels. This architecture also contains $\lceil \lg(n) \rceil - 2$ buffer control counters $bcc_1, \ldots, bcc_{\lceil \lg(n) \rceil - 1}$ and the selection function φ to select the correct floating-point-core outputs to be written to the buffers.

The algorithm for the architecture is shown in Fig. 8. In the algorithm, out_n is the tree level at which the final result may be found and equals $\lceil \lg(n) \rceil$. $singleton_i$ denotes whether or not the last value at level i is a singleton. If $i = 0$, $singleton_i = n \bmod 2$; otherwise, $singleton_i = \lfloor n/2^{i+1} \rfloor \bmod 2$ ($i = 1, \ldots, \lceil \lg(n) \rceil$). $last_i = 1$ when a value entering the floating-point cores is the last for that set of inputs to enter the floating-point cores at level i of the expression tree and $last_i = 0$ otherwise.

2) Proof of Correctness: In the advanced case, the scheduling of buffer $1, \ldots$, buffer $\lceil \lg(n) \rceil - 1$ is the same as the scheduling of buffer $0, \ldots$, buffer $\lg(n) - 2$ in the basic case. Thus, the correctness of the algorithm and architecture can be proven in the same way as in Section III-A2. In this section, we only present two issues whose proof cannot be explicitly derived from the basic case.

Lemma 5: A buffer cannot grow without bound once we begin reading values from that buffer.

Proof: This is obvious for buffer 0. For buffer i ($i = 1, \ldots, \lceil \lg(n) \rceil - 1$), since we may have a singleton case, either one or two values are read every 2^i cycles. Thus, $1 \times 1/2^i \leq \mathrm{rate}_{\mathrm{read}} \leq 2 \times 1/2^i = 1/2^{i-1}$. During n cycles, at most $\lceil n/2^{i+1} \rceil$ values are written to buffer i. Thus, $\mathrm{rate}_{\mathrm{write}} \leq \lceil n/2^{i+1} \rceil / n$. If $n \geq 2^{i+1}$, then $\lceil n/2^{i+1} \rceil / n \leq 1/2^{i+1} + 1/n \leq 1/2^i$. If $2^i < n < 2^{i+1}$, then $\lceil n/2^{i+1} \rceil / n = 1/n < 1/2^i$. We know that $\mathrm{rate}_{\mathrm{write}} \leq 1/2^i$, for all $n > 2^i$.

In our algorithm, any arithmetic expression that reaches level i must have more than 2^i inputs. Otherwise, the expression has $out_n \leq i$ and the evaluation is complete before level i. Therefore, we have $\mathrm{rate}_{\mathrm{write}} \leq \mathrm{rate}_{\mathrm{read}}$. We thus prove that we cannot have unbounded growth once we start reading from the buffer. □

Theorem 6: The latency of the algorithm is at most $3n + (\alpha - 1)\lceil \lg(n) \rceil - 3$ cycles.

Proof: It takes n cycles for all the inputs to arrive. The last value enters buffer 0 at cycle n, and then traverses the first set of floating-point cores. Then it enters buffer 1, one of the second set of floating-point cores, and so on until it enters buffer $\lceil \lg(n) \rceil - 1$ and then one of the second set of floating-point cores. Clearly it goes through the floating-point cores $\lceil \lg(n) \rceil$ times and at buffer i, it waits for at most $2^i - 1$ cycles before being read by the floating-point cores ($i = 1, \ldots, \lceil \lg(n) \rceil - 1$). Therefore, the latency of the algorithm is at most $n + \alpha \lceil \lg(n) \rceil + \sum_{i=1}^{\lceil \lg(n) \rceil - 1} (2^i - 1) \leq n + \alpha \lceil \lg(n) \rceil + 2n - 2 - \lceil \lg(n) \rceil - 1 = 3n + (\alpha - 1)\lceil \lg(n) \rceil - 3$ cycles. □

C. Properties of the Designs

In this section, we detail some important properties of our designs for both the basic case and the advanced case. The first of these is that, as desired, at most two floating-point cores for each type of operator in the expression are present in the designs. Each floating-point core in the designs is being used efficiently; each floating-point core takes new inputs as often as possible. This is an important property because, due to their large areas, the floating-point cores dominate the area of expression evaluation circuits. This property would not be as notable if large buffers were necessary or the time complexity for the expression evaluation increased to more than the optimal, which is $\Theta(n)$ since it takes at least n cycles for all of the inputs to arrive.

Second, only $\Theta(\lg(n))$ buffer space is necessary for the designs. This is a small amount of space and leaves most of the FPGA's on-chip memory for other tasks implemented on the device. Despite this small buffer size, as shown in Theorem 4 and Theorem 6, the latency remains low. The designs also achieve the highest possible throughput for inputs arriving sequentially: an output every n cycles once the floating-point-core pipelines are full.

Another property of the designs is that they place no restriction on the number of pipeline stages in the floating-point cores. In developing the designs, we assumed that the floating-point cores all had the same number of pipeline stages. In practice, this is not always the case. To compensate, shift registers can be added to the floating-point cores with fewer stages so that their pipeline latencies match that of the floating-point core with the most stages. α will be set to the number of pipeline stages of the floating-point core with the longest pipeline latency. While adding shift registers does increase the area of the architecture, this increase will be insignificant when compared to the size of the floating-point cores. Importantly, no pipeline stalls are introduced.

The designs also allow for the inclusion of the extra pipeline stages before or after the floating-point cores, if necessary. This extra delay can be captured by increasing the value of α by the number of added pipeline stages. For example, if the critical path of the design is between the output of the floating-point cores and the input to memory, an extra pipeline stage (or more than one stage) can be added to the path. It is only necessary to increase the value of α by the number of added stages. This property adds scalability to the algorithm.

{counters}
if the first pipeline is first used **then**
  $C = 0$
  for $i = 1$ to $\lceil \lg(n) \rceil$ - 1 in parallel **do**
    $bcc_i = 0$
  end for
else
  $C = C + 1$
end if
{buffers}
write input port to buffer 0
$bcc_0 = bcc_0 + 1$
if $out_n \leq 1$ **then**
  write $\varphi(0, bcc_0)$ to external memory
else
  write $\varphi(0, bcc_0)$ to buffer 1
  $bcc_1 = bcc_1 + 1$
end if
for $i = 1$ to $\lceil \lg(n) \rceil - 2$ **do**
  if $(C - \alpha)_{\langle i \rangle} = 2^i - 1$ **then**
    if $out_n \leq i + 1$ **then**
      write $\varphi(i, bcc_i)$ to external memory
    else
      write $\varphi(i + 1, bcc_i)$ to buffer $i + 1$
      $bcc_{i+1} = bcc_{i+1} + 1$
    end if
  end if
end for

{floating-point cores}
if $last_0 = 1$ **and** $singleton_0 = 1$ **then**
  read 1 value from buffer 0
else if 2 values are in buffer 0 **then**
  read 2 values from buffer 0
end if
for all $s \in$ the first set of operators in parallel **do**
  enter the read values into floating-point core s
end for
for $i = 1$ to $\lg(n) - 1$ **do**
  if $C_{\langle i \rangle} = 2^i - 1$ **then**
    if $last_i = 1$ **and** $singleton_i = 1$ **then**
      read 1 value from buffer i
    else
      read 2 values from buffer i
    end if
  end if
  for all $s \in$ the second set of operators in parallel **do**
    enter the read values into floating-point core s
  end for
end for

Fig. 8. Algorithm for the advanced case.

Last, multiple expressions can be evaluated using the same hardware without reconfiguration. If there are m expressions to be evaluated and each expression has the same number of inputs, the only difference in evaluating the expressions is the φ function. The outputs of the φ function can be stored in memory, so only the memory needs to be changed to change the expression being evaluated. If the expressions to be evaluated have different numbers of inputs, this can be handled as well. The sizes for the memories and counters should be set for the expression with the largest number of inputs. As long as the φ function is set correctly for a given expression, its result can be read from the evaluation circuit at the appropriate time.

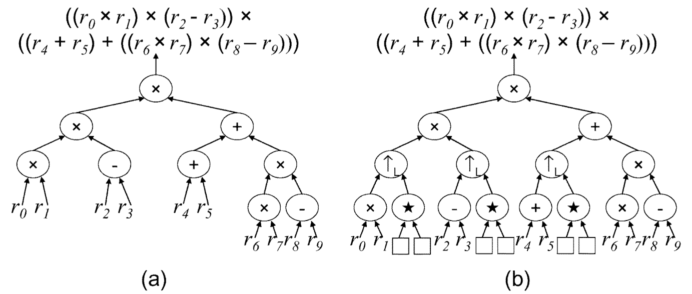

Fig. 9. (a) Expression tree before padding. (b) Expression tree after padding ($\star$ represents any operation. $\square$ represents a padding input, $\uparrow_{\mathrm{L}}$ represents the operation of passing the left input to the next level).

D. General Expressions

Thus far, we have only considered certain classes of arithmetic expressions. Despite this, the proposed algorithms and architectures can be applied to a wide variety of general expressions. For example, some expressions involve nonbinary operators. As a preprocessing step, the expression trees for such expressions can be transformed into binary trees as discussed in Section II-A.

After such transformations, the resulting binary expression tree still may not comply with the requirements of the basic case or the advanced case. In this case, the input source can pad the inputs to the expression evaluation circuit until the expressions meet the requirement. For example, the expression tree in Fig. 9(a) could be transformed into the expression tree in Fig. 9(b). Doing such padding reduces the throughput. If h is the number of levels in the tree and n is the number of inputs, then the throughput is reduced by a factor of $2^h/n$. If the binary tree is balanced—that is, if the heights of the subtrees of any node differ by only a small amount—the impact on the throughput of the algorithm and architecture is small. For example, for an expression such as that in Fig. 9, where the subtrees of the root node differ in height by one, the reduction in throughput is at most a factor of 2. As discussed in Section II-A, various methods exist that can reduce the height of the expression trees and transform the trees into more balanced ones. Through padding, our designs can be applied to general expressions.

IV. Experimental Results

We analyze the performance of the proposed solution to the basic case for 64 inputs and 1024 inputs as the number of types of floating-point operators ranges from a single type of operator to four types of operators. We think of these operators as add, subtract, multiply, and divide. However, because our goal is to study the performance of our design rather than the performance of individual floating-point cores, we use the same floating-point core in all cases, repeating it as many times as is

necessary. In our implementation, we use a floating-point multiplier core. For example, in the case of four types of operators, rather than having a floating-point adder, subtracter, multiplier, and divider, we use four floating-point multipliers.

The multiplier, as well as the other floating-point cores used for this paper, are described in [8]. In summary, the multiplier has ten pipeline stages and requires 935 slices of area and 16 of the embedded 18×18 multipliers. It takes inputs in IEEE-standard-754 double-precision format and uses a round-toward-zero rounding mode. This multiplier is sufficient for the purpose of this illustration. Any pipelined multiplier, even a fixed-point one, could be used with our architecture. If a more or a less complex multiplier is required, the performance of the proposed design will change accordingly.

We coded the proposed architecture and algorithm for the basic case in VHDL. With the Xilinx Virtex-II Pro XC2VP20 as our target device, we synthesized the design with Synplicity Synplify Pro 8.1 and placed-and-routed the result with Xilinx PAR 7.1. Since buffer 0 is written to every clock cycle, we implemented it as two registers. In order to avoid large multiplexers at the inputs to the floating-point cores, we did not implement the other buffers separately. Instead, we combined them into two embedded RAMs, one for the "left" inputs to the floating-point cores and the other for the "right" inputs. Also, to simplify the addressing logic, we allowed each buffer level in the RAM to have four words instead of just three. With this scheme, we only need to multiplex between the registers of buffer 0 and the RAM for the other buffer levels. The address logic takes care of choosing between the buffers at levels 1 through $\lg(n) - 1$. We do still, however, require multiplexers to select which floating-point core's output should be written into the buffers.

Fig. 10 shows the effect on the frequency and the area of the implementation as the number of operators goes from one to four. For both input sizes, the area increases roughly linearly with the number of types of operators. Note that the area for $n = 1024$ is not much larger than the area for $n = 64$. In each case, as the number of types of operators increases, the frequency decreases. This makes sense because the amount of control logic necessary increases as the number of operators increases. A curious case is the relatively small decrease in frequency between one and two types of operators when $n = 64$. In each case except for $n = 64$ with one type of operator, the critical path is in the control logic dealing with writing to the buffers. However, when $n = 64$ and there is only one type of operator, the critical path is in the floating-point multiplier. That is, in this case, the design cannot achieve a higher frequency because it has reached the limit of the floating-point multiplier, rather than reaching the limit of the control logic. Because of these different limiting factors, with $n = 64$, there is not a great difference in frequency between when there is one type of operator and when there are two.

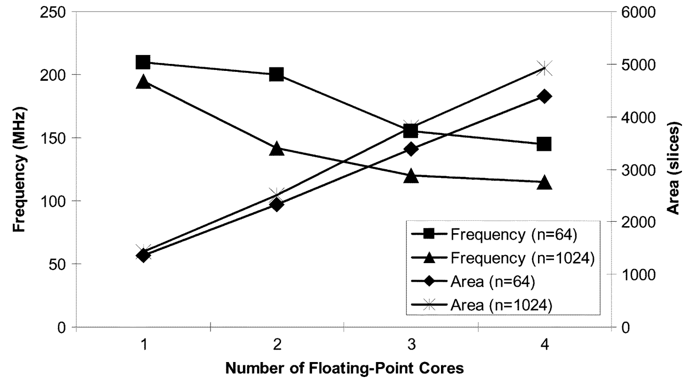

Fig. 10. Change in frequency and area as number of types of operators goes from 1 to 4.

TABLE I
THROUGHPUT-TO-AREA RATIO OF THE PROPOSED ARCHITECTURE AND THE COMPACTED-TREE ARCHITECTURE. VALUES ARE IN THOUSAND RESULTS PER SLICE PER SECOND (HIGHER VALUE IS BETTER)

	Number of Types of Operators			
	1	2	3	4
proposed	2.40	1.34	0.71	0.52
compacted-tree	0.40	0.27	0.18	0.13

TABLE II
AREA-LATENCY PRODUCT OF THE PROPOSED ARCHITECTURE AND THE COMPACTED-TREE ARCHITECTURE. VALUES ARE IN SLICES $\times \mu$s PER RESULT (LOWER VALUE IS BETTER)

	Number of Types of Operators			
	1	2	3	4
proposed	1115	1994	3742	5180
compacted-tree	4866	7053	10880	14924

A. Comparisons

Here, we compare the proposed solutions with the more straightforward compacted tree architecture (as in Fig. 4) and with compiler-generated designs. As the use of floating-point cores on FPGAs is new, we are unaware of any other relevant work with which to compare the proposed solutions.

1) Compacted Tree: In Tables I and II, the implementation of the proposed solution to the basic case is compared with an implementation of the compacted-tree architecture. Table I shows the throughput-to-area ratio of each of the architectures when $n = 64$ and the number of types of operators varies from one to four. Clearly, the proposed solution has a much higher throughput-to-area ratio; the area used is being used much more efficiently. Area–latency product is another common efficiency metric for circuits. Table II shows the area– latency product of the two architectures and algorithms. While the compacted-tree algorithm requires fewer cycles to compute a result than the proposed design, its area-latency product is much higher than that of the proposed design. This shows that, while computing results only slightly more slowly, the proposed design is much more area-efficient than the compacted tree design.

2) Compiler-Generated Designs: We now compare the proposed designs to designs for the same expressions generated by a high-level language compiler. The compiler we choose to compare against is the compiler in SRC Computers' Carte programming environment, which is one of the state-of-the-art compilers presently available [18]. This compiler allows hardware designs to be described in the C programming language and includes some extra macros for data movement and special functions. The compiler generates an executable for SRC's MAPstation, which has two microprocessors and two Xilinx Virtex-II XC2V6000 FPGAs. We focus only on the FPGA part and all results presented in this section are for designs targeted to the Virtex-II XC2V6000. Additionally, all designs run at 100

MHz as this is a requirement for the SRC 6 MAPstation. In these comparisons, the designs generated by the SRC compiler use SRC's floating-point cores while our design use the floating-point cores from [8].

One function for which a hardware macro is provided is floating-point accumulation. SRC's floating-point accumulation macro is called from within a pipelined loop. If multiple sets of numbers are to be accumulated, the entire pipeline must be flushed to finish one accumulation before any numbers from the next set can enter the pipeline to begin the next accumulation. Accumulation is a special case of the proposed designs in which there is only one type of operator: addition. The proposed designs do not require that the pipeline be flushed between sets of numbers. The drawback to the proposed designs compared to the SRC accumulator is that the number of inputs to the accumulation must be known in advance whereas the SRC accumulator can accumulate conditionally and take any number of inputs.

To accumulate 64 and 1024 inputs, the proposed design for the basic case requires 1474 and 1589 slices, respectively, while the SRC accumulator requires 2002 slices in each case. Thus, the proposed design uses less area while not requiring any flushing of the pipeline. If the number of inputs were not a power of two, the proposed design for the advanced case would be used, requiring about twice as much area because of the need for two floating-point adders. Still, though, no pipeline flushing would be required.

We also compare the area of the proposed design for the basic case in evaluating arbitrary expressions to the designs generated by the compiler. As examples, we look at an 8-input expression and a 64-input expression. For an 8-input expression with 2 adds, 1 subtract, and 4 multiplys, the proposed design requires 3378 slices while the design generated by the compiler requires 6823 slices. The trouble is that the compiler is limited to creating designs in which each floating-point operation has its own floating-point core. It cannot generate optimized designs like that developed here. For an 8-input expression, this is not too bad, as the generated design easily fits on the target FPGA. However, as the number of inputs increases, the FPGA quickly fills up. A 64-input expression with 18 adds, 17 subtracts, 17 multiplys, and 11 divides requires only 6738 slices with the proposed design, but requires 56 565 slices in the compiler-generated design. 56 565 slices is 67% more slices than are present in the target FPGA, so the expression cannot be placed and routed.

B. Using the Proposed Design in a Kernel

We briefly describe how the proposed solution could be used as part of a kernel. In [9], we propose an FPGA-based architecture for sparse matrix-vector multiplication. In the architecture, the compacted-tree architecture with floating-point adders is used to accumulate intermediate results. Compared with a highly optimized program for sparse matrix-vector multiply on an Itanium 2 system, the architecture achieves 1.1× to 30× speedup depending upon the sparsity structure of the matrix. With the algorithm and architecture proposed in this paper, we can further reduce the area requirement of the architecture in [9], thus making it possible to handle larger matrices and use a smaller FPGA to achieve the same performance.

V. Related Work

A. Parallel Arithmetic Expression Evaluation

The arithmetic expression evaluation problem has been studied widely in the parallel computing community. For example, [3] and [10], present parallel algorithms for the problem that assume the PRAM model in which there are many processors, all with uniform access to a shared memory. In [1], the algorithm in [3] is adapted to be practical for SMP machines. [11], on the other hand, solves the problem for the processor arrays with reconfigurable bus system model in which computation is performed by cells connected to a grid-shaped reconfigurable bus system. Each of these techniques employs *tree contraction* [4]. In tree contraction, a rooted, binary tree is systematically reduced from consisting of many nodes to consisting of a single node—the root. These methods for arithmetic expression evaluation require $O(\lg(n))$ time.

These existing works in parallel computing assume that a processor is capable of performing any operation. Additionally, it is assumed that a processor counts as a unit of computing resource. However, in FPGA-based designs, each operation has its own operator that occupies certain hardware resources. Therefore, we are not concerned with the number of processors and communication time. Instead, we aim to minimize the number of operators while achieving a high throughput.

Reference [12] discusses area-efficient designs for parallel polynomial evaluation on FPGAs. The goal of this work is to minimize the hardware requirements of the evaluator by using the minimal number of operators for each type of operation. Since they only consider fixed-point arithmetic, each operation takes one clock cycle to complete. For a polynomial with degree $n, \Theta(n)$ operators are required. We focus on floating-point expression evaluation and the floating-point cores are all pipelined. Moreover, the number of operators in our proposed solution is independent of the input size.

B. Pipeline Synthesis

Datapath pipeline synthesis has been studied by many researchers. Many times, such as in [13]–[15], pipelining is performed only at the task level and the individual operators are not pipelined. More closely related to our work are works such as [16] and [17] in which the individual operators and the overall tasks are pipelined.

A scheduling algorithm for two-level pipelining, where both the task and each of the operators are pipelined, is proposed in [16]. The scheduling problem is formulated as an integer linear programming problem for minimization of the total execution time. The running time of the algorithm depends on the number of nodes in the dataflow graph, the number of operators, and the number of pipeline stages in each operator.

Reference [17] proposes a design methodology that automatically generates FPGA circuits that meet a given throughput constraint at minimal area cost. Two different scheduling algorithms are used in the methodology, both of which combine module selection with resource sharing during pipeline synthesis. The library elements used in the synthesized designs are prepipelined

and their numbers of pipeline stages are taken into account in the algorithms.

In all these works on pipeline synthesis, design-space exploration has to be performed for each individual dataflow graph. The output of the works is dependent on the dataflow graph and the selected modules. On the other hand, our proposed architectures and algorithms can be used for all expressions with little modification. Moreover, our designs are independent of the implementations of the floating-point cores.

VI. Conclusion

In this paper, we have presented algorithms and architectures that facilitate the area-efficient evaluation of arithmetic expressions using deeply pipelined floating-point cores. We have demonstrated the correctness of the algorithms and that the performance achieved by the proposed designs is far superior to that of other techniques. Beyond area efficiency, the proposed designs have several benefits, including high throughput, a low memory requirement, and the ability to evaluate multiple expressions without hardware reconfiguration. Because they only receive one input per clock cycle, they also have a low input/output bandwidth requirement.

In the future, we will pursue the use of the designs as part of the acceleration of large floating-point application kernels. For example, we would like to apply the algorithm and architecture for the advanced case to tasks from molecular dynamics. We will also investigate changing the algorithm to account for "skinny" trees that cannot be transformed. These are just some of the many opportunities for future research that are facilitated by the proposed designs. There are many others in many different fields, including scientific computing, cognition, and graph theory.

References

[1] D. Bader, S. Sreshta, and N. Weisse-Bernstein, "Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs)," in *Proc. 9th Int. Conf. High Perform. Comput.*, 2002, pp. 63–75.

[2] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *J. Assoc. Comput. Mach.*, vol. 21, no. 2, pp. 201–206, Apr. 1974.

[3] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *Proc. 26th IEEE Symp. Foundations Comput. Sci.*, 1985, pp. 478–489.

[4] J. JáJá, *An Introduction to Parallel Algorithms*. Reading, MA: Addison-Wesley, 1992.

[5] D. Kuck and Y. Muraoka, "Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity," *Acta Inform.*, vol. 3, no. 3, pp. 203–216, Sep. 1974.

[6] D. Kuck and K. Maruyama, "Time bounds on the parallel evaluation of arithmetic expressions," *SIAM J. Comput.*, vol. 4, no. 2, pp. 147–162, Jun. 1975.

[7] A. V. Kozlov and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *Proc. 1994 ACM/IEEE Conf. Supercomput.*, 1994, pp. 320–329.

[8] G. Govindu, R. Scrofano, and V. K. Prasanna, "A library of parameterizable floating-point cores for FPGAs and their application to scientific computing," in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms*, 2005, pp. 137–148.

[9] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. 13th ACM Int. Symp. Field-Program. Gate Arrays*, 2005, pp. 63–74.

[10] R. Cole and U. Vishkin, "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time," *Algorithmica*, vol. 3, pp. 329–346, Mar. 1988.

[11] B. Pradeep and C. S. R. Murthy, "Parallel arithmetic expression evaluation on reconfigurable meshes," *Comp. Lang.*, vol. 20, no. 4, pp. 267–277, Nov. 1994.

[12] M. Wojko and H. ElGindy, "On determining polynomial evaluation structures for FPGA based custom computing machines," in *Proc. 4th Australasian Comput. Arch. Conf.*, 1999, pp. 11–22.

[13] N. Park and A. Parker, "Sehwa: A program for synthesis of pipelines," in *Proc. 23rd Des. Autom. Conf.*, 1986, pp. 454–460.

[14] P. Paulin and J. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proc. 24th Des. Autom. Conf.*, 1987, pp. 195–202.

[15] R. Jain, A. Parker, and N. Park, "Module selection for pipelined synthesis," in *Proc. 25th Design Autom. Conf.*, 1988, pp. 542–547.

[16] C. Chen and M. Moricz, "Data path scheduling for two-level pipelining," in *Proc. 28th Des. Autom. Conf.*, 1991, pp. 603–606.

[17] W. Sun, M. Wirthlin, and S. Neuendorffer, "Combining module selection and resource sharing for efficient FPGA pipeline synthesis," in *Proc. 14th ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2006, pp. 179–188.

[18] D. S. Poznanovic, "Application development on the SRC Computers, Inc. Systems," in *Proc. IEEE Int. Parallel Distrib. Symp.*, 2005, p. 78a.

Ronald Scrofano (M'06) received the B.S. degree in computer science and mathematics from California Lutheran University, Thousand Oaks, in 2001, and the M.S. and Ph.D. degrees in computer science from the University of Southern California, Los Angeles, in 2003 and 2006, respectively.

Since Fall 2007, he has been with the Computer Systems Research Department, The Aerospace Corporation, El Segundo, CA. His research interests include reconfigurable computing, supercomputing applications, and emerging architectures.

Dr. Scrofano is a member of the IEEE Computer Society and the ACM.

Ling Zhuo (M'07) received the B.A. degree in engineering from Tsinghua University, Beijing, China, in 2000, the M.S. degree in electrical engineering and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, in 2005 and 2007, respectively, and the M.Phil. degree in computer science from the University of Hong Kong, Hong Kong, in 2002.

She is currently a Postdoctoral Research Associate with the Ming Hsieh Department of Electrical Engineering, University of Southern California. Her research interests include algorithm design for high-performance scientific applications on reconfigurable computing systems, parallel and distributed computing, and computer architecture.

Viktor K. Prasanna (F'96) is Charles Lee Powell Professor of Engineering with the Ming Hsieh Department of Electrical Engineering and Professor with the Computer Science Department, the University of Southern California (USC), Los Angeles. He served as the Division Director for the Computer Engineering Division during 1994–1998. His research interests include parallel and distributed systems including networked sensor systems, embedded systems, configurable architectures, and high performance computing. He has published extensively and consulted for industries in these areas.

Dr. Prasanna was a recipient of the 2005 Okawa Foundation Grant. He is an associate member of the Center for Applied Mathematical Sciences (CAMS), USC. He has served on the organizing committees of several international meetings in VLSI computations, parallel computation, and high performance computing. He is the Steering Co-chair of the International Parallel and Distributed Processing Symposium [merged IEEE International Parallel Processing Symposium (IPPS) and the Symposium on Parallel and Distributed Processing (SPDP)] and is the Steering Chair of the International Conference on High Performance Computing (HiPC). He has served on the editorial boards of the *Journal of Parallel and Distributed Computing*, Proceedings of the IEEE, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, and IEEE Transactions on Parallel and Distributed Systems. He served as the Editor-In-Chief of the IEEE Transactions on Computers during 2003–2006. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. (See http://ceng.usc.edu/~prasanna for further details.)