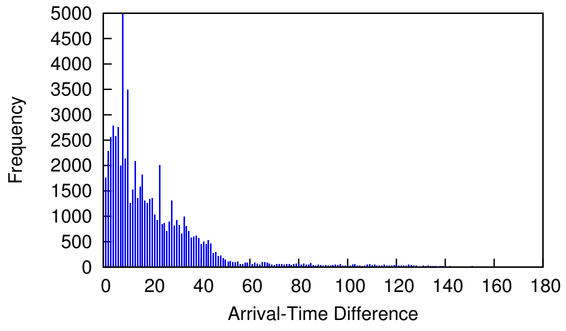
Criticality-driven Token Dataflow Optimizations for FPGA-based Sparse LU Factorization

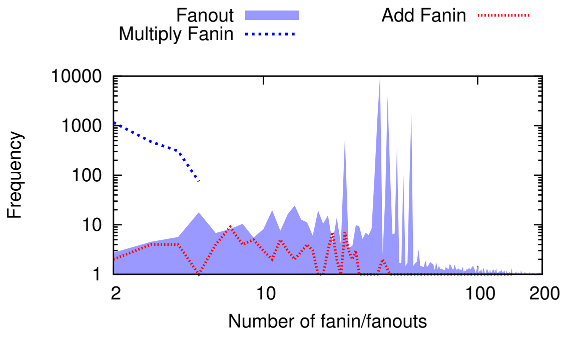
Removed for blind review
Removed for blind review

***Abstract*—**

Performance of FPGA-based token dataflow architectures is often limited by the long tail distribution of parallelism in the compute paths of dataflow graphs. This is known to limit speedup of dataflow processing of Sparse LU factorization to only 3–10× over CPUs. In this paper, we show how to overcome these limitations by exploiting criticality information along compute paths; both statically during graph pre-processing and dynamically at runtime. We statically restructure the high-fanin dataflow chains using a technique inspired by Huffman encoding where we provide faster routes for late arriving inputs as predicted through our timing models. We also perform a fanout decomposition and selective node replication in order to distribute serialization costs across multiple PEs. This static restructuring overhead is small; roughly the cost of a single iteration, and is amortized across 1000s of LU iterations at runtime. Additionally, we modify the dataflow firing rule in hardware to prefer critical nodes when multiple nodes are ready for dataflow evaluation. We compute this criticality offline through a one-time slack analysis and implement this in hardware at virtually no cost through a trivial address encoding ordered by criticality. For dataflow graphs extracted for sparse LU factorization, we demonstrate up to 2.5× (mean 1.21×) improvement when using the static pre-processing alone, a 2.4× (mean 1.17×) improvement when using only runtime optimizations alone while an overall 2.9× (mean 1.39×) improvement when both static and runtime optimizations are enabled across a range of benchmark problems.



(a) Arrival-time variation (t_{last} - t_{first})



(b) Fanin/Fanout Distribution

I. Introduction

FPGA-based token dataflow architectures are an increasingly important design choice for accelerating many hard computational problems where parallelism is sparse, and irregular. In these circumstances, a raw unrolled dataflow graph exposes all possible parallelism in the computation in its purest form. The dataflow architectures allow asynchronous, decoupled evaluation of parallelism in irregular graphs without the need for clumsy parallelization hacks. The dataflow graph execution proceeds using a simple dataflow firing rule where a node is fired when all its inputs are received. Sparse LU factorization is one such representative engineering application that is notoriously hard to parallelize and is considered a challenging problem for conventional processors. It is a well-known compute bottleneck in fields such as circuit simulation [5], computational fluid dynamics [3], machine learning, bioinformatics, among many others. When the sparse matrices are fixed, we can extract its unique dataflow graph for LU factorization. Hardware-assisted token dataflow acceleration of Sparse LU implementations (*e.g.* [6], [10]) can deliver non-trivial speedups of 3–10× over CPU-based solvers. However, this is achieved with barely 10% of dataflow operator utilization due to the long tail distribution of parallelism in the graph.

Dataflow graphs can often have a few nodes with large fanin or fanout counts. As we scale system sizes, high fanin/fanout nodes quickly become the performance bottleneck. Furthermore a dataflow graph can have multiple concurrent parallel paths from inputs to outputs but certain paths will be more timing critical than others. If we schedule evaluation without regard to timing criticality of compute paths, we will delay completion thereby lowering performance. By careful ordering of operations, we can deliver additional speedups for FPGA-based token dataflow processing. Based on these insights, we propose three optimizations in this paper:

- *Fanin reassociation*: We could trivially decompose the high fanin nodes using associative property into balanced reduction trees that assume uniform input arrival time. However, that is a generous assumption as observed for addition operator nodes shown in Figure 1(a) (for the `bomhof2` benchmark). This shows a high 180-cycle gap between the fastest and slowest inputs. We need to build associative fanin trees that account for this arrival time.

- *Fanout decomposition/replication*: Typically fanout nodes are processed in sequence in the dataflow PE, resulting in serialization bottlenecks for large fanouts (see Figure 1(b)). We need to reduce this overhead by (1) distributing fanout serialization across multiple PEs, as well as (2) prioritizing evaluation of edges that must travel further in the system.
- *Criticality-driven dataflow firing*: At a given evaluation cycle, there may be several nodes active in a processing element (PE). If we simply use the first-in-first-out (FIFO) order, we may unintentionally ignore evaluation of more timing critical compute paths. Hence we need to perform criticality-based selection of node evaluation in each PE at runtime.

The key contributions in this paper are:

- Design of a dataflow compiler that performs arrival-time aware reassociation of high fanin nodes as well as fanout decomposition/replication in the dataflow graphs.
- Redesign of dataflow PE hardware to support criticality-driven dynamic scheduling during runtime.
- Quantification of performance of the dataflow compiler and hardware modification on sparse matrix benchmarks selected from the circuit simulation domain.

II. Background

A. Token Dataflow Architecture

Token Dataflow architectures were the subject of academic studies in the early 1990s *e.g.* [7], [2], [4], [**?**]. However, due to the emergence of the killer microprocessors, these designs and ideas were largely relegated to academic projects. At an abstract level, the dataflow architecture is composed of PEs connected by switched network fabric. Computation on this architecture is organized as a sequence of “token” communication along graph dependency edges and subsequent “dataflow firing” at the graph nodes. Each PE has local memory blocks that are used to store portions of the dataflow graph for localized processing. Each PE is capable of performing logic and/or arithmetic operations on each node of the graph based on a dataflow firing rule. Under this rule, each node is allowed to independently and asynchronously compute when it has all inputs ready. Dependencies between nodes are routed through the packet-switched token communication network. For FPGA-based systems, dataflow processing offers an unique opportunity to deliver a reprogrammable and scalable computing substrate that can be tailored to different applications. In this paper, we consider a customized heterogeneous token dataflow architecture optimized for sparse LU factorization as the vehicle for our experiments and optimizations.

B. Sparse LU Factorization

In many numerical problems, we are required to solve a set of linear equations expressed as $A\vec{x} = \vec{b}$ in matrix-vector notation. Matrix A is often a highly sparse matrix that stays structurally unchanged when working with real-world applications. For example, in circuit simulation, each circuit component is only connected to a few neighboring elements, thereby resulting in very localized non-zero patterns when we represent the circuit as a matrix. The hardware design we consider in this paper is based on the KLU solver [1], which is a software package for solving sparse matrix systems. KLU does a one-time pre-ordering step that fixes non-zero locations in the matrix, hence, allowing us to keep the dataflow and memory structure static throughout an iterative process. This step is especially suitable for parallel hardware-assisted solvers [6], as there is no need to recompute the dataflow graph and do dynamic memory allocation in each iterative step. At the heart of the KLU solver is the Gilbert-Peierels (GP) algorithm (Listing 1). The GP algorithm is responsible for generating the L & U factors for the input matrix A. In [6], the authors unroll the for-loop in the GP algorithm to generate giant dataflow graphs that represent the GP compute flow. However, a front-solve (line 4) must be carried out in each step of the for-loop, which becomes a compute bottleneck as the lower-triangular (L) matrix is iteratively built in each for-loop iteration. In [8], the authors target this front-solve by doing a one-time recursive depth-limited substitution and reassociation to further expose any available parallelism. In this paper, we test our dataflow optimizations on the substituted and reassociated dataflow graphs and simulate performance on a heterogeneous token dataflow hardware architecture.

Algorithm 1: Gilbert-Peierls

```
Data: sparse matrix A
Result: factors L & U
1 L = I;
2 for i=1:N do
3     b = A(: , i);
4     x = L\b;
5     U(1:i , i) = x(1:i);
6     L(i+1:N , i) = x(i+1:N) / U(i , i);
7 end
```

C. Token Dataflow Architecture for Sparse LU Factorization

For our intended scenario, the front-solve in sparse LU factorization operates in single-precision floating-point arithmetic. The numerical calculations are mostly multiplies and adds with a few divides. For our heterogeneous dataflow design shown in Figure 1, we customize the ALU functions handled by each PE to reflect this distribution. We also customize the communication network by adding a faster multi-hop channel between the add PEs to allow critical dependencies to be routed faster.

III. Criticality-driven dataflow optimizations

In this section, we explain the static and dynamic criticality-aware optimizations performed in our dataflow framework.

A. Overview and Motivation

Performance of token dataflow architectures is often limited by the long tail distribution of parallelism in the dataflow graph. Within these constraints, performance is exacerbated

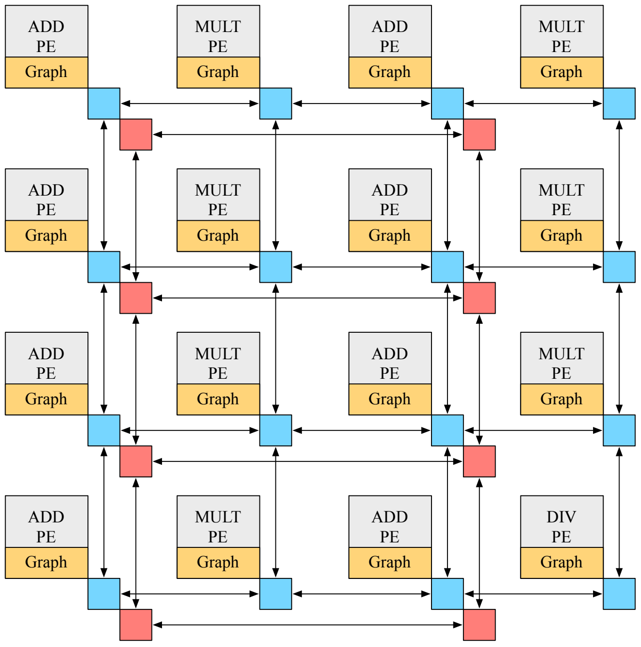
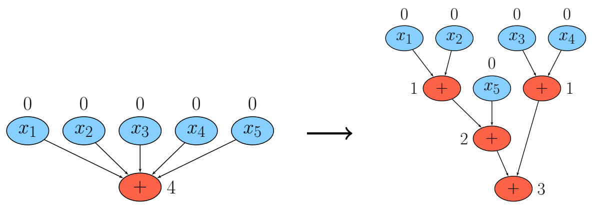


Fig. 1: Heterogeneous Token Dataflow architecture (add:mult = 1:1, two NoC channels)

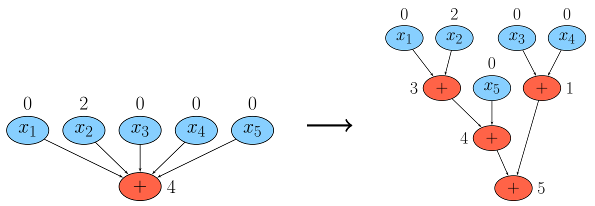
by the oblivious processing of high-fanin and high-fanout nodes without awareness of criticality. Wherever possible, we can exploit arithmetic associativity to transform the high-fanin nodes into a reduction tree to distribute the serialization bottleneck across multiple PEs. Additionally, we can create copy nodes to decompose high-fanout nodes to achieve a similar performance improvement. However, if we approach this problem without considering timing criticality, instead of improving performance, we can actually make it worse.

Fanin: For high-fanin nodes, if the inputs arrive at the same times, the fanin inputs can be reassociated into a reduction tree naively, as shown in Figure 2(a). However, as shown in Figure 1(a), arrival times of inputs can vary significantly, which then suffer additional delays in the reduction tree stages resulting in an overall increase in cycle count. This defeats the purpose of reassociation. This effect is illustrated in a simple example shown in Figure 2(b) where we now require 5 cycles to finish evaluation with oblivious reassociation as opposed to the 4 cycles in the raw fanin case. Hence, we propose a static one-time pre-processing step that generates reduction trees in a manner that exploits predicted arrival time information (How do we do this? See Section III-B). Figure 2(c) shows the desired tree construction for the same example with the same varying input arrival times now finishing in 3 cycles thereby beating both the oblivious reassociation as well as original fanin chain.

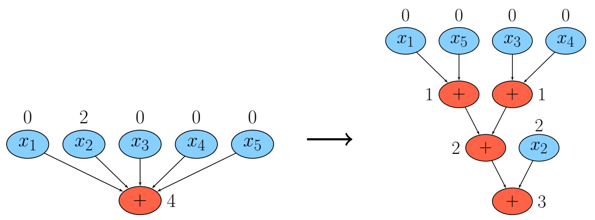
Fanout: For high-fanout nodes, serialization of packet transmissions can limit performance as only one fanout can be serviced every cycle. To tackle this issue, we propose a fanout decomposition scheme as demonstrated in Figure 5 (See Section III-C). We use a similar strategy for constant input



(a) Uniform input arrival time: latency savings of 1 from simple reassociation



(b) Non-uniform input arrival time: latency increase of 1 from blind reassociation



(c) Non-uniform input arrival time: latency savings of 1 from intelligent reassociation

Fig. 2: Reassociation of high fanin nodes into a reduction tree (latency of two-input add operation = 1 cycle)

nodes with large fanout, where instead of creating a fanout tree, we perform locality-aware node replication, which is a cheap memory tradeoff for improved performance.

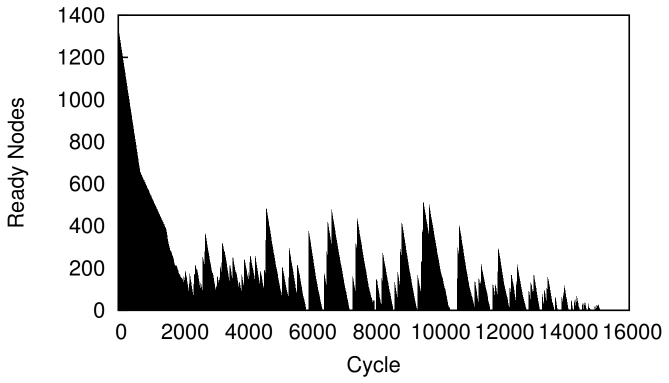
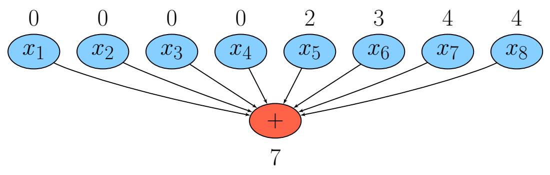
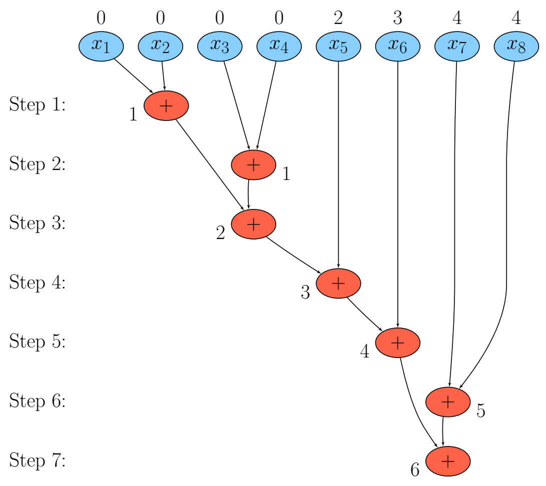


Fig. 3: Number of nodes ready (max across all PEs) in any given cycle (single iteration of large graph in `bomhof2`)

Node Selection: While static dataflow graph optimizations can help to optimize compute order in the dataflow graph, multiple dataflow nodes may be ready for evaluation at runtime. Figure 3 shows the number of nodes ready (max across all PEs) in any given cycle for one of the large dataflow graphs from `bomhof2` benchmark. How do we pick between these



(a) High Fanin Node with varying input arrival time



(b) Huffman-styled reassociation

Fig. 4: Huffman-styled reassociation on a high fanin node with varying input arrival times (Latency Savings = 1)

Algorithm 2: Huffman-styled fanin reassociation

```
Data: Priority Queue $V$, Input fanins $f_1, f_2, ..., f_n$ labeled with
      ASAP timing $t_i$
Result: Timing-optimized reduction tree
foreach $f_i$ in $f_1, f_2, f_3 \ldots, f_n$ do
    $t_i$ = $f_i$.getASAP();
    $f_i$.setHuffmanTime($t_i$);
    $V$.push($f_i$);
end
while $V$.size() != 1 do
    inp1 = $V$.pop();
    inp2 = $V$.pop();
    op_node = createOperator();
    op_node.connectInputs(inp1,inp2);
    t1 = inp1.getHuffmanTime();
    t2 = inp2.getHuffmanTime();
    t3 = MAX(t1, t2) + 1;
    op_node.setHuffmanTime(t3);
    $V$.push(op_node);
end
```

ready nodes? To optimize overall runtime, we should pick nodes along the critical path that allow fastest progress towards termination. A naive implementation would require the use of expensive priority queues in hardware, but we develop a simple solution that eliminates this need (See Section III-D for more details).

B. Criticality-Aware Reassociation (Static)

To implement criticality-aware reassociation of computation represented in a dataflow graph, we draw inspiration from the Huffman-encoding algorithm [**?**]. In Huffman encoding, a tree is constructed based on the probability of occurrence of each input symbol being encoded. While constructing the tree, the symbol probabilities are accumulated incrementally and each stage has balanced probabilities. We adapt this algorithm to use arrival times instead of symbol probability when constructing the fanin tree for a high-fanin node. We compute the arrival times based purely in the static structure of the dataflow graph through a simple ASAP analysis. This is a lower-bound estimate of the time when the node will be available for downstream computations as we do not model network congestion costs and queuing of ready nodes in the PEs. Despite the approximation, there is a strong correlation between cycle count and predicted ASAP latencies, such that a reduction in ASAP critical path reflects as an improvement in performance on hardware. For simplicity we consider a operation latency of 1 for both add and multiply nodes. In Figure 4, we shows an example of arity-2 fanin tree construction based on this adapted Huffman scheme. Listing 2 shows the pseudo-code for implementing this reassociation scheme. This code has an asymptotic complexity of $N \times log(N)$ where N is typically in the low 100s enabling rapid execution. We can assert that this Huffman-styled reassociation scheme produces superior results based on the following deductions:

Lemma 1: Doing a blind reassociation (without regard to the input arrival times) on node n with m-fanins, each with a statically-predicted ASAP arrival time of t_i where $i = 1 : m$, the worst node-ready time, t_{n1}, is shown in Equation 1 where $log_2(m)$ is the height of a balanced fanin tree.

$$t_{n1} = max_{i=1:m}(t_i) + \lceil log_2(m) \rceil \tag{1}$$

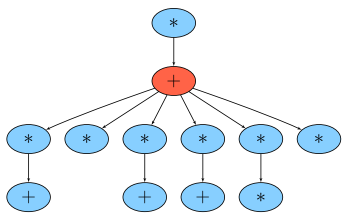
Lemma 2: For a node n with m-fanins, its worst-case node-ready time, t_{n2}, is shown in Equation 2. Here n_{max} is the number of input nodes with the identical maximum arrival time of $max(t_n)$.

$$t_{n2} = max_{i=1:m}(t_i) + \lceil log_2(2 * n_{max})/2 \rceil \tag{2}$$

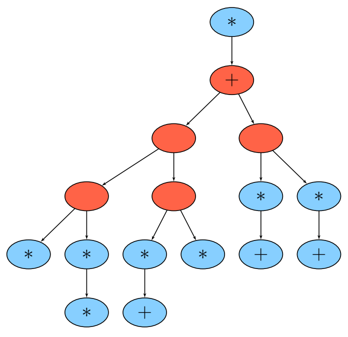
When comparing, Equation 1 with Equation 2, we can see that if we ignore arrival time information, the worst-case fanin path will always suffer a $log_2(m)$ internal tree delay. The arrival time aware technique will only delay the last arriving input by the shortest sized tree required to accommodate the fanin subset ($n_{max} \leq m$) arriving at that particular time. When $n_{max} = 1$, we will only delay the last arriving input by a single stage (accounts for the odd-looking $\lceil log_2(2*n_{max})/2 \rceil$ term). When all fanin arrival times are the same, both techniques generate the same balanced fanin trees.

C. Fanout Decomposition and Replication (Static)

To implement fanout decomposition, we first define two control parameters: threshold (f_t) and arity (f_a). If the fanout size of a node in our input dataflow graph is greater than f_t,



(a) Undecomposed



(b) Arity-2 Decomposition

Fig. 5: Fanout Decomposition Example

we perform fanout decomposition on that node. The decomposition is carried out such that arity of decomposed fanout tree is not greater than f_a, *i.e.* each node in the decomposed tree has no more than f_t fanouts. Under these constraints, we decompose the fanouts in the most balanced way possible, such that the fanouts are distributed across the new copy nodes (new red nodes in 5(b)) as evenly as possible. Selecting values for f_t and f_a could be potentially complex – for example, we could design a dynamic threshold/arity selection scheme based on graph properties, PE configuration and/or placement information. In this paper, we observed that a configuration of $f_t = 16$ and $f_a = 4$ delivered consistently good speedups over all tested dataflow graphs.

D. Criticality-Driven Packet Scheduling (Runtime)

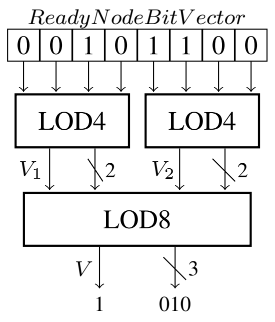


Fig. 6: Leading-one detector in each PE to process nodes based on criticality (left-to-right)

To implement a criticality-driven runtime optimization scheme, we performance slack analysis on the dataflow graph through a cheap, one-time ASAP and ALAP analyses. As a result, each node is labeled with a statically-determined earliest ready time t_{ASAP}, and a latest ready time t_{ALAP}. Based on these values, we find the *slack* available at each node, which can be computed using the expression in Equation 3.

$$S_n = t_{n,ALAP} - t_{n,ASAP} \quad (3)$$
$$C_n = (w_t - S_n)/w_t \quad (4)$$

Finally, in order to identify the critical path, we compute the *criticality* value at each node using Equation 4, where w_t is the worst-case node-ready time determined from ASAP analysis of the dataflow graph. The value of C_n lies between 0 and 1. A C_n closer to 1 indicates that the node lies along the critical path and should be given priority scheduling over nodes with smaller C_n values. Note that thus far, we have only added a one-time static dataflow graph labeling scheme which is cheap to implement during the static compiler phase.

Once we have computed *criticality* for each node in the dataflow graph, we compute node addresses in each PE based on this criticality. This idea eliminates the need to allocate expensive priority queues in hardware. We use node address directly when choosing which node to fire. We use a leading-one detector (LOD) to find the node to process in each step. An LOD is a well-studied circuit (*e.g.* [**?**]) for detecting the position of the largest non-zero MSB in an input n-bit vector. For long vectors, the LOD is designed in a hierarchical manner. Figure 6 shows a high-level example of how an LOD is implemented for an input 8-bit vector. The detector is designed with LOD4–LOD8 hierarchy, and hence, the speed of this design example is proportional to $log_4(n)$ stages. The final LOD8 returns a 3-bit address value for the detected leading one, and a $Valid$ output bit. This hierarchical design can be optimized to meet different area/performance goals (*e.g.* LOD2–LOD4–LOD8 hierarchy is possible). Furthermore, to support a large number of nodes in a PE without increasing node selection costs, we fix the size of the LOD and reuse the detector multiple times. This is possible as most nodes have large fanout and we only need to detect the next node after those edges are processed.

IV. Methodology

In this section, we detail our compilation flow and comment on our hardware design characteristics.

A. Dataflow Compilation Flow

For quantifying the performance limits of our dataflow hardware and compiler, we extract dataflow graphs for sparse LU matrices. Our matrix pre-processors converts input matrices from circuit simulation domain represented in the Matrix Market (*.mtx*) format into corresponding dataflow graphs. Our dataflow compiler applies fanin and fanout transformations as discussed earlier in Section III. We also modify the addressing logic for nodes to automatically include criticality. We show the sequence of applying these transformations in Figure 7. We quantify any speedups observed with reference to the baseline performance in [6], where neither static nor runtime

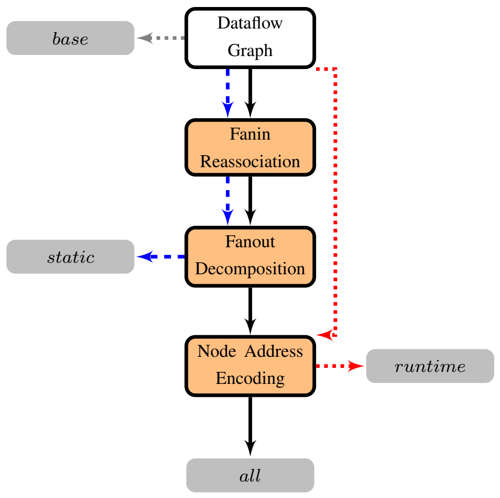


Fig. 7: Dataflow Compiler Flow-Chart

optimizations were used. Our flow is currently supporting sparse LU factorization graphs, but it is general and applicable to other domains beyond circuit simulation where applications can be characterized by large, irregular dataflow graphs.

B. Dataflow Hardware Design

We target the Xilinx Virtex-6 SX475T FPGA device similar to the one used in [6]. This limits the largest dataflow processor we can accommodate on this system to 12x12 (144 PEs). The switching latencies are calibrated to meet the target 250MHz design frequency. Our design can optimally support dataflow graphs that can fit the entire data structure into the on-chip BRAM memory blocks. We approximate the largest dataflow graph size, n_{max}, that can fit onto a total memory size of M using the following expression:

$$n_{max} = \frac{M}{c_{id} + p_c * c_{fp} + (1 - p_c) * c_{fl} + p_e * c_e} \quad (5)$$

where p_c is the percentage of nodes being constant nodes (observed to be $\approx$0.3), p_e is the ratio of edges:nodes (observed to be $\approx$1.25), c_{id} is the cost of storing each node ID (=2.125 bytes), c_{fp} is the cost of storing a single-precision floating-point number (=4 bytes), c_{fl} is the cost of storing dataflow flags for each node (=0.25 bytes), and c_e is the cost for storing single edge information for packet construction (=3.125 bytes). Using Equation 5, the $\approx$ 37Mb of on-chip memory can fit dataflow graphs with up to approximately 5 million nodes (single-precision). While most benchmarks studied in this paper meet this requirement, some of the larger benchmarks may not be able to completely fit inside the on-chip memory. Our performance models include the same external memory loading costs as in [6].

V. Results

In this section, we present our results and offer a brief discussion and future outlook on the observed trends. We test a total of 22 benchmarks, extracted from the circuit simulation domain. The benchmarks range in size/sparsity from hundreds to tens of thousands of non-zeros. The graph properties of dataflow graphs extracted from the input matrices are tabulated in Table I.

TABLE I: Benchmark Graph Properties

Benchm.	**Rows**	**Sp.**	**Graph Properties**					
			Nodes	**Edges**	**Const.**	**Adds**	**Mults**	**Crit. Path**
bomhof1	2,624	0.5%	1.9m	2.6m	628k	575k	711k	24k
bomhof2	4,510	0.1%	6.1m	8.4m	1.6m	1.4m	2.4m	49k
bomhof3	12,127	0.03%	760k	959k	280k	203k	277k	48k
simucad	4,875	0.3%	6.6m	8.8m	2.2m	1.9m	2.5m	75k
s27	189	3.3%	5.4k	5.7k	2.6k	0.9k	1.9k	1k
s208	1,296	0.5%	116k	137k	47k	22k	46k	12k
s298	1,801	0.4%	220k	267k	87k	48k	85k	16k
s344	1,992	0.3%	126k	145k	54k	22k	50k	15k
s349	2,017	0.3%	129k	147k	55k	23k	51k	13k
s382	2,219	0.3%	287k	351k	111k	57k	119k	20k
s444	2,409	0.3%	354k	435k	136k	80k	137k	28k
s386	2,487	0.3%	286k	340k	116k	56k	113k	21k
s510	2,621	0.3%	609k	746k	207k	119k	226k	26k
s526n	3,154	0.2%	544k	669k	209k	124k	210k	27k
s526	3,159	0.2%	550k	674k	212k	124k	212k	27k
s641	3,740	0.2%	839k	1.1m	301k	177k	352k	50k
s713	4,040	0.2%	810k	1.0m	302k	178k	323k	56k
s820	4,625	0.2%	657k	796k	259k	143k	254k	42k
s832	4,715	0.2%	938k	1.2m	355k	218k	361k	47k
s953	4,872	0.2%	4.0m	5.3m	1.1m	841k	1.6m	76k
s1196	6,604	0.1%	5.9m	7.5m	1.8m	1.4m	2.0m	142k
s1238	6,899	0.1%	12.8m	16.8m	3.7m	3.2m	4.6m	200k

Sp. = Sparsity

A. Understanding Performance Improvement

We tabulate the performance observed when evaluating our benchmark dataflow graphs with different static/runtime optimizations in Table II. We note speedups between 0.9–2.9× when considering both optimizations. For small benchmarks `s27`, `s349` and `s382`, we observe a slowdown of 0.8–0.9× as the graphs are too small to benefit from restructuring. In some cases, individually applying static and dynamic optimizations actually causes a slowdown of 0.7–0.9× but when considered together, we observe a nominal speedup of 1.1×. For large dataflow graphs, speedups are proportionally larger as noted in Figure 8. We only start observing speedups for benchmarks larger than $\approx$100K nodes.

In Figure 9, we show the effect of scaling PEs on performance of a select 6 benchmarks. We note a linear improvement in performance at small PE counts with a saturation effect at PE counts above 64 for most of the cases. Certain benchmarks like `s641` and `s953` do not scale particularly well due to limited parallelism in the input itself (long critical paths).

TABLE II: Benchmark Cycles and Speedups

Benchm.	**Cycles and Speedup**						
	BASE	STATIC		RUNTIME		ALL	
bomhof1	2.1m	854k	2.5×	886k	2.4×	718k	2.9×
bomhof2	2.7m	1.4m	1.9×	1.5m	1.8×	1.2m	2.3×
bomhof3	1.2m	1.1m	1.1×	1.2m	1.0×	1.0m	1.2×
simucad	6.7m	2.8m	2.4×	2.9m	2.3×	2.3m	2.9×
s27	17k	23k	0.7×	24k	0.7×	22k	0.8×
s208	229k	270k	0.8×	276k	0.8×	245k	0.9×
s298	388k	400k	1.0×	404k	1.0×	356k	1.1×
s344	265k	328k	0.8×	338k	0.8×	302k	0.9×
s349	254k	303k	0.8×	308k	0.8×	279k	0.9×
s382	470k	513k	0.9×	506k	0.9×	462k	1.0×
s444	648k	688k	0.9×	708k	0.9×	612k	1.1×
s386	487k	516k	0.9×	514k	0.9×	458k	1.1×
s510	707k	652k	1.1×	625k	1.1×	581k	1.2×
s526	807k	740k	1.1×	749k	1.1×	667k	1.2×
s526n	809k	739k	1.1×	749k	1.1×	666k	1.2×
s641	1.2m	1.3m	0.9×	1.3m	0.9×	1.1m	1.1×
s713	1.3m	1.4m	0.9×	1.4m	1.0×	1.2m	1.1×
s820	1.1m	1.1m	0.9×	1.1m	0.9×	970k	1.1×
s832	1.4m	1.3m	1.0×	1.3m	1.1×	1.2m	1.2×
s953	2.8m	2.0m	1.4×	2.1m	1.3×	1.8m	1.6×
s1196	5.2m	3.5m	1.5×	3.7m	1.4×	3.2m	1.6×
s1238	10.2m	5.1m	2.0×	5.7m	1.8×	4.7m	2.2×
GEOMEAN			**1.21×**		**1.17×**		**1.39×**

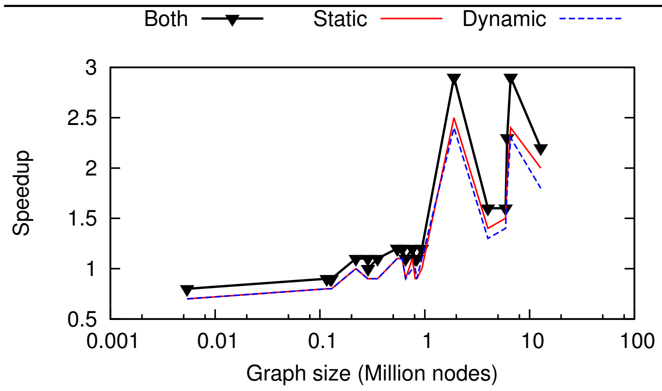


Fig. 8: Correlation between size of dataflow graph and speedup

B. *Static vs. Dynamic Optimizations*

We note that both optimization styles have significant impact on performance. In certain cases, for medium-sized benchmarks shown in Figure 8, it is important to apply both optimization to achieve speedup. The cumulative effect of both these transformations is not always multiplicative as static optimizations limit the damage caused by oblivious dynamic scheduling. Nevertheless, there is notable improvement in performance when both optimizations operate together.

We now attempt to understand the individual impact of these optimizations. In Figure 10, we note that in a vast majority in cases for the bomhof2 benchmark, the use of Huffman-style reassociation improves performance by as much as 40% across all internal iterations. A few outliers suggest noise effects due to placement effects or dynamic ready node ordering. In Figure 11, we show the speedups observed from only enabling fanout decomposition. Only sufficiently large graphs with high fanout nodes benefit from this optimization, and hence, only graphs with >10k nodes are shown in the plot. The speedups observed are about 1–10% with few outlier cases. These outlier cases are due to placement effects (minor slowdowns) or unoptimal choice of f_t and f_a parameters (significant slowdowns). In this work, we have adopted generic, best-across-all values of f_t and f_a for fanout decomposition, and there is certainly room for improvement for dynamic parameter selection for fanout decomposition. In Figure 12, we quantify the effect of applying smarter dynamic node selection at runtime. In this case, performance improved by as much as 30% across all iterations. Generally speaking, we observe that the static optimizations translate to slightly better speedup than dynamic optimizations. Static optimizations have access to the full dataflow graph structure when making decisions over graph transformations whereas dynamic optimizations in hardware only have access to instantaneous criticality information.

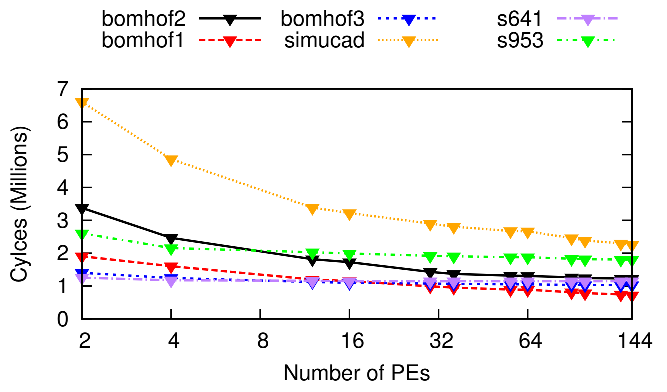


Fig. 9: Cycles vs PEs scaling trends for several benchmarks when all optimizations are enabled

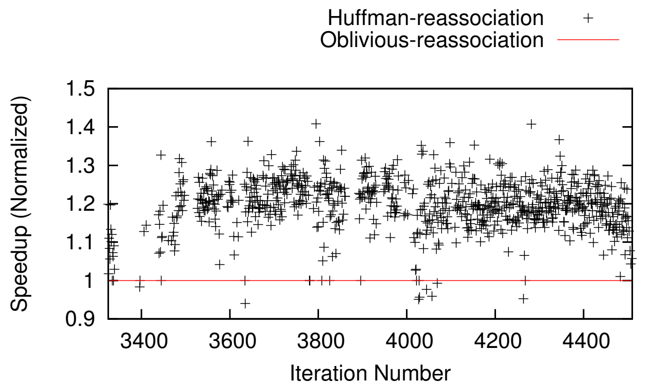


Fig. 10: Speedups observed with only Huffman-styled STATIC optimization (bomhof2)

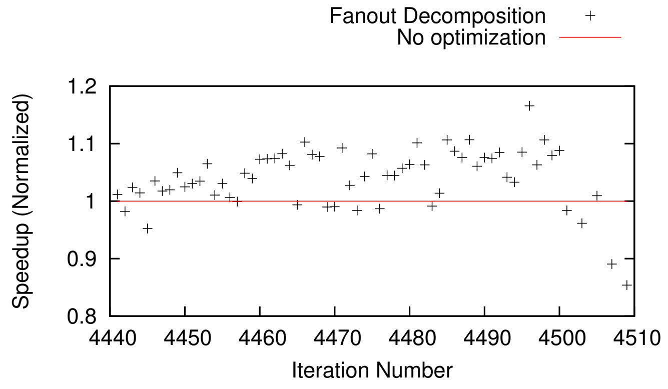


Fig. 11: Speedups from only fanout decomposition ($f_t = 16, f_a = 4$)

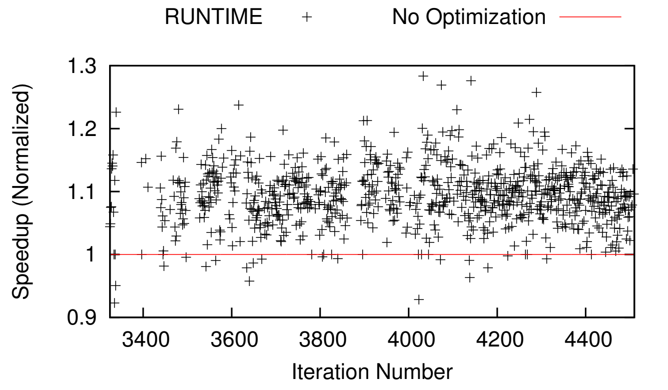


Fig. 12: Speedups observed with only RUNTIME optimization (`bomhof2`)

C. Time spent by packets in communication network

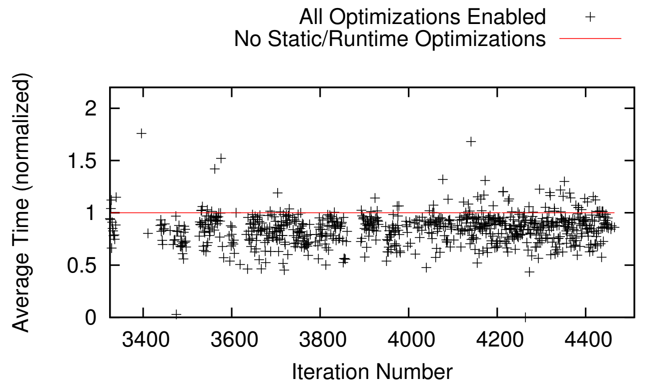


Fig. 13: Average time (normalized) spent by packets in network for BOTH optimizations (`bomhof2`)

The time (number of cycles) spent in the communication network by packets is a strong indicator of the congestion effects in the communication network. Figure 13 shows the drop in the average time spent in the communication channels by packets for different front-solve iterations when we enable both the dataflow optimizations proposed in this paper. Again, the few outliers that slow down due to placement effects or variations due to dynamic node selection.

VI. Conclusions

In this paper, we show how to improve performance of FPGA-based token dataflow architectures through suitable static and runtime dataflow optimization strategies. We show how to achieve an additional speedup of up to 2.9× (mean 1.39×) on top of existing performance of parallel dataflow hardware. Our static optimizations focus on reordering fanin computations in a dataflow graph based on ASAP timing models and decomposing/replicating high fanout operator/constant nodes to reduce serialization delays. Our runtime optimization focuses on developing an intelligent dynamic packet-scheduling scheme without introducing area overheads. All the optimizations take advantage of statically-computed metrics such as *ASAP/ALAP times* and *criticality*, which are negligible one-time costs in highly iterative applications (*e.g.* SPICE).

VII. Future Work

We intend to extend our compiler to handle dataflow graphs from other domains beyond circuit simulation. There is further scope for improvement by exploiting locality information when performing fanin reassociation. We can also modify the placement pass to include two iterations; once at the start to extract preliminary locality hints and a final iteration that requires placement of the transformed fanin-fanout graph.

References

[1] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, A Direct Sparse Solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, Sept. 2010.

[2] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3(4):126–132, Dec. 1974.

[3] J. H. Ferziger and M. Perić. *Computational methods for fluid dynamics*, volume 3. Springer Berlin, 1996.

[4] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, Jan. 1985.

[5] N. Kapre. *SPICE2–A Spatial Parallel Architecture for Accelerating the SPICE Circuit Simulator*. PhD thesis, California Institute of Technology, Pasadena, 2010.

[6] N. Kapre and A. DeHon. Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs. In *Field-Programmable Tech.*, 2010.

[7] G. Papadopoulos. Monsoon: a dataflow computing architecture suitable for intelligent control. *Intelligent Control, 1990. Proceedings., 5th IEEE International Symposium on*, 1990.

[8] Siddhartha and N. Kapre. Breaking Sequential Dependencies in FPGA-based Sparse LU Factorization. In *The International Conference on Field Programmable Logic and Applications 2014*, pages 1–4, Sept. 2014.

[9] Siddhartha and N. Kapre. Heterogeneous Dataflow Architectures for FPGA-based Sparse LU Factorization. In *FPL '14: Proceedings of the 2014 22nd IEEE Symposium on Field Programmable Custom Computing Machines*, pages 1–4, Mar. 2014.

[10] X. Wang and S. G. Ziavras. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. *Concurrency and Computation: Practice and Experience*, 2004.