# Limits of Statically-Scheduled Token Dataflow Processing

Nachiket Kapre, Siddhartha
School of Computer Engineering
Nanyang Technological University
Singapore 639798
*nachiket@ieee.org*

***Abstract*—**
**FPGA-based token dataflow processing has been shown to accelerate hard-to-parallelize problems exhibiting irregular dataflow parallelism by as much as an order of magnitude when compared to conventional compute organizations. However, when the structure of the dataflow computation is known upfront, either at compile time or at the start of execution, we can employ static scheduling techniques to further improve performance and enhance compute density of the dataflow hardware. In this paper, we identify the costs and performance trends of both static and dynamic scheduling approaches when considering hardware acceleration of SPICE device equations and Sparse LU factorization in circuit graphs. While the experiments are limited to a case study, the hardware design and dataflow compiler are general and can be extended to other problems and instances where dataflow computing may be applicable. With this study, we hope to develop a quantitative basis for the design of a hybrid dataflow architecture that combines both static and dynamic scheduling techniques. We observe a performance benefit of 2–4× and a resource utilization saving of 2–3× in favor of statically scheduled hardware.**

## I. Introduction

Performance and power requirements of a computation mapped to physical hardware are dependent on application characteristics and the underlying physical costs of communication, memory storage and computation. One important characteristic that affects implementation costs is application parallelism. Mainstream multi-cores and GPUs are able to manage coarse-grained, regular and embarrassingly data-parallel problems quite easily. However, solutions for managing irregular and fine-grained forms of parallelism are still broadly elusive. Dataflow may be one possible solution. Dataflow architectures, which were pioneered in the early 1990s, provide an alternative form of parallel organization but have been largely relegated to academic curiosity. The key feature of dataflow machines is the ability to exploit large amounts of physical hardware resources to deliver scalable performance for certain applications through asynchronous decoupled operation and communication of dependencies using "tokens". These architectures support distributed dataflow triggering to launch parallel computations in a manner that does not require expensive synchronization (*e.g.* program counter) and tolerates variable communication and memory delays in the system. This execution mechanism is a good match for irregular and fine-grained parallel applications. In these architectures, computations are described as dataflow graphs where the nodes represent computation (*e.g.* arithmetic and/or logical operations) and edges represent dependencies between operations (*e.g.* communication). Despite there favorable characteristics, dataflow architectures were unable to compete with the density and scaling advantages made possible by microprocessors (*e.g.* Intel CPUs). However, there is a need to revisit the dataflow architecture models once again as we discover that a broad class of important hard-to-parallelize problems defy mapping to multi-cores and GPUs.

A canonical view of dataflow computing involves tokens and dynamic scheduling to permit distributed, decoupled evaluation of parallelism in the problem. This view simplifies hardware and compiler design and allows scalability to newer technologies and capacities. However, there is an overhead associated with supporting this flexibility. In this paper, we investigate the cost/performance gap between traditional dynamically scheduled dataflow and statically scheduled alternatives. We expect static scheduling to offer improved performance due to better opportunities for global optimization of the processor design while, at the same time, potentially using fewer resources. We recognize that static scheduling may not be possible in all contexts particularly where the dataflow graph structure is unknown until much later in an execution lifetime. Additionally, there is no clear classification for statically scheduled dataflow in the broader dataflow taxonomy (See Figure 1 for our attempt). However, it is important to first understand the extent of cost/performance loss and identify research directions that may close this gap. We use the SPICE circuit simulator as a case study and an FPGA-based implementation platform for exploring these questions regarding dataflow organizations.

The key contributions of this paper include:

- RTL design and cycle-accurate simulator for evaluation of simple statically-scheduled and dynamically-triggered processors implemented as 2D mesh on a Xilinx FPGA.
- Development of a dataflow graph pre-processing engine for communication-aware distribution of dataflow graphs on the 2D mesh.
- Development of a static scheduler that includes a greedy XY router that optimizes and schedules dataflow graphs

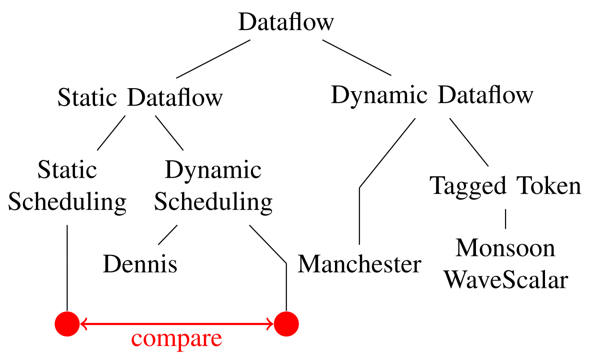


Fig. 1: Dataflow Taxonomy

for a 2D-mesh organized static dataflow processor.

- Experimental framework to characterize and compare performance of dataflow graphs extracted from the SPICE circuit simulator across benchmark devices and circuits.

## II. Background

### A. Dataflow Organization Taxonomy

As shown in Figure 1, dataflow architectures can be broadly categorized into *static* and *dynamic* models. We derive this taxonomy through analysis of the excellent review of different dataflow architectures presented in [5]. Static dataflow architectures permit only a single instance of the graph to be active in hardware at any given time, greatly simplifying the hardware design [3]. Dynamic dataflow machines [4], [12] allow multiple instances of the same dataflow graph to be active at the same time, complicating the token tracking hardware but generalizing the applicability. Over time, variations on these ideas have given us newer dataflow derivatives. Some of the recent proposals [1], [14] involve ideas that handle instruction placement and issue through a combination of static and dynamic methods. In this work, we adopt a different approach that investigates the opportunity for static placement as well as scheduling (analogous to instruction issue). A key inspiration for this project is [10] which explores static scheduling of dataflow graphs for DSP applications while focusing on reduced complexity of dataflow triggering. Our work expands the scheduler to include communication over a shared network into the model and actually quantifies the performance and cost gap for a case study. Similar to EDGE [1] and WaveScalar [14] projects, we perform static instruction placement where our instructions are specialized node operations customized to the application being accelerated. However, depending on our scheduling model, we issue our instructions either dynamically or statically. For the benchmark problems we consider (explained in Section IV), we perform if-mux conversion to eliminate the impact of divergent control flow on dataflow evaluation and consequently we do not have to consider the impact of branch divergence in our static schedule. This important transformation allows static scheduling of code blocks larger than the traditional basic block (of hyperblocks [2]), but potentially at the expense of

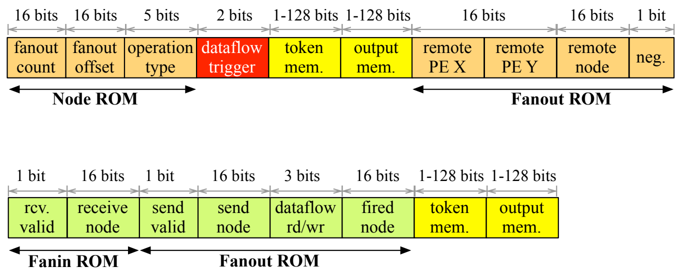


Fig. 4: Memory Layout

extra work. Furthermore, our static scheduling is more general than VLIW scheduling as it supports non-loop oriented code as well. The static scheduler allows us to determine the precise schedule length (runtime) of the dataflow evaluation at compile time. For dynamically scheduled hardware we have to rely on a cycle-accurate simulation to report performance.

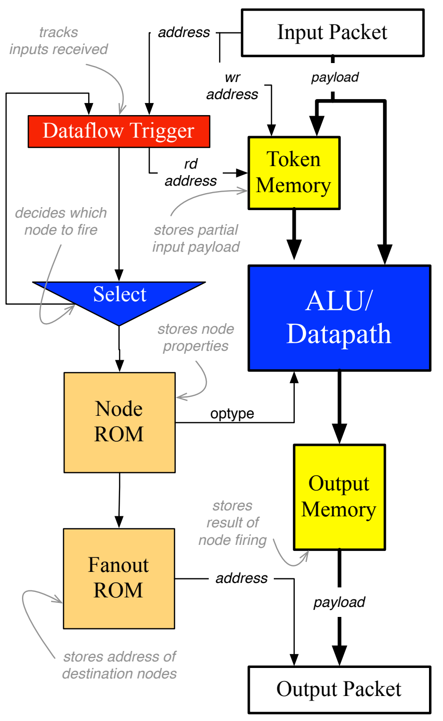
Computations that are exclusively scheduled through static techniques are broader in scope than pure dataflow. We see this research as a stepping stone towards the development of hybrid dataflow architectures that combine both scheduling styles together. In this context, WASMII [11] is an early dataflow overlay architecture for FPGAs that uses the dynamic scheduling approach for token handling while using a statically configured FPGA context for programming the dataflow operator. There is also some recent interest in developing FPGA PE hardware [13] for supporting the EDGE ISA [1].

## III. Idea – Hardware Configurations for Dataflow

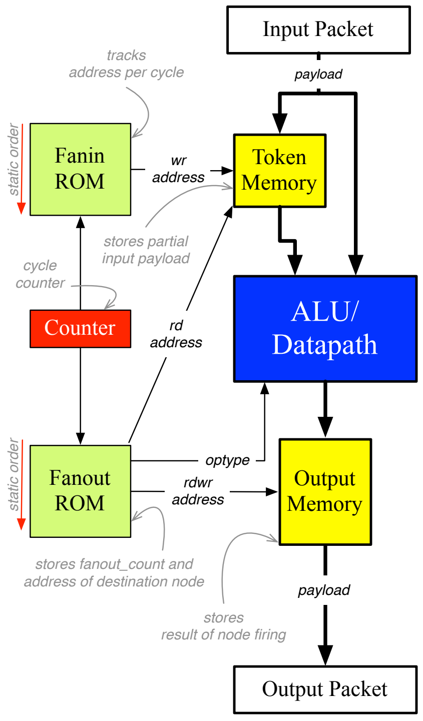
In this section, we describe the two competing designs that can exploit dataflow parallelism using static and dynamic scheduling. We identify the conditions under which they offer characteristic performance and consume specific resources. The dataflow processor is organized as a 2D mesh of dataflow processing elements (PEs) interconnected with a token routing 2D mesh fabric. Designs of both the PE and router, which we discuss next, are specialized for the scheduling model used.

### A. Dataflow PE Architecture

*Dynamically-Scheduled PE*: As shown in Figure 2a, the design of the *canonical* dataflow processor can be composed of two coupled hardware blocks: token triggering logic, and the token generator. The token triggering logic tracks the number of inputs received by the operator and issues the dataflow operation when the trigger condition is activated *i.e.* expected token count is observed. Inputs to a dataflow node are stored in a Token memory while the node waits for the other input(s). After a dataflow operation has been issued, it is committed to the Output Memory and the node is marked as ready for dispatch. Under the dynamic scheduling model, the token generator must select a node for processing among a potential set of multiple ready nodes. The dataflow operation results of the chosen node are tokenized and dispatched to the appropriate destinations. The stored Fanout memory represents the graph

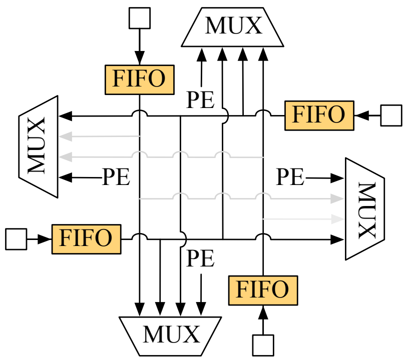


(a) Token Dataflow

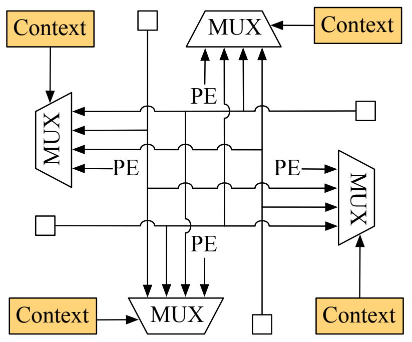


(b) Static Scheduling

Fig. 2: Dataflow Micro-architectures using Dynamically-scheduled and Statically-scheduled Hardware Engines



(a) Token Dataflow



(b) Static Scheduling

Fig. 3: Router designs for Dynamically-scheduled and Statically-scheduled Hardware Engines

structure that encodes the required addressing information to dispatch tokens.

*Statically-Scheduled Dataflow PE*: We represent the organization of the statically-scheduled implementation of the same design in Figure 2b. We observe several structural simplifications that can lower implementation cost and improve frequency. The complete processor execution is known a priori – the processor merely executes a set of pre-determined steps for both input and output handling. Thus, we store the execution-schedule in ROMs that are loaded at the start, thereby eliminating any need to make dataflow triggering decisions at runtime. Significant savings can also be obtained from the storage costs associated with representing the dataflow graph in memory. As shown in Figure 4, we can see that static processing only requires send and receive memories with associated memory access controls per cycle. For large systems with low schedule lengths, this can represent a significant saving in memory as we will discover later in Section VI.

## *B. Dataflow Router Architecture*

In Figure 3, we show the design of the dataflow routing switches used in both cases. When the dataflow graph is evaluated dynamically, the packet format includes both data and address information that is required for the packet to route over the network to its destination. For statically-scheduled implementation, we can eliminate the need to package addressing information in the packet. This is possible as we can statically identify the packet dispatch times and exact path taken through the network at compile time. In the dynamically-routed switch, we need to buffer the packets and use address information to route. While this makes the network flexible, there is an added memory and performance overhead. Instead, the statically-routed switch requires storage of a statically-identified context (*e.g.* instructions for the switch muxes). This limits the network to only support a particular dataflow graph at a given time, but greatly simplifies the design and allows high frequency implementations. There is an additional limitation of the largest schedule length that may be supported but with cheap high-density memories, this limit is high enough for our case study.

### C. Hardware Utilization

We generate RTL for the dataflow graphs using cycle-accurate, C++ descriptions of the dataflow engines (as seen earlier in Figure 2) compiled using a high-level synthesis (HLS) toolflow. We compose a parallel dataflow system in a 2D mesh by connecting the processing elements with packet-switched and time-multiplexed switches depending on the scheduling model. These switches are also described using high-level C++ code. We use the Xilinx Vivado HLS compiler 2013.4 and Xilinx Vivado 2013.4 to generate RTL and compile bitstreams for the Zynq Z7010 FPGA chip. Our HLS-generated clock delays are particularly severe for dynamic PEs and switches (≈10–15ns) compared to static counterparts (≈4–6ns). We continue to investigate HLS coding patterns that may reduce this gap. The key variables that define hardware usage and performance for the dataflow architectures are (1) datapath precision and operation counts (for both architectures), and (2) schedule length (only for the statically-scheduled architecture). The size of the parallel dataflow system defines the address width contained in the packet being routed over the dataflow token routing network. For our experiments we assume a maximum system size of 16×16 (256 PEs) and an 8-bit packet address. The size of subgraph accommodated in the PE defines the memory requirements for storing (1) variables at nodes and edges during dataflow evaluation (for both architectures), and (2) scheduling context (for the statically-scheduled architecture). For the purpose of comparing synthesis results fairly, we fix the nodes and edges per PE to be 1024. When comparing resource costs, we measure LUTs (*i.e.* logic gates), FFs (*i.e.* registers) and BRAMs (*i.e.* embedded memory blocks) utilization on the FPGA. For Xilinx FPGAs, LUTs can be reprogrammed as small 32–64b RAMs which we use to store the scheduling context.

In Figure 6, we show the effect of increasing datapath precision on processor and routing resources. As expected, utilization increases linearly with increasing precision. However, we observe a larger relative cost in the switching fabric due to buffering requirements of packet-switched networks that can sometimes be tricky to implement correctly on FPGAs using

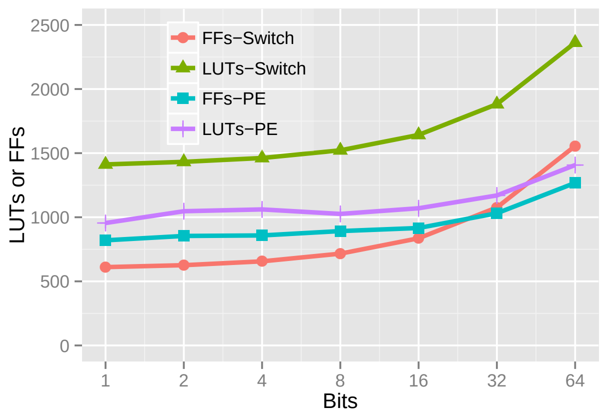


Fig. 6: FPGA Hardware Costs for Dynamic Scheduling

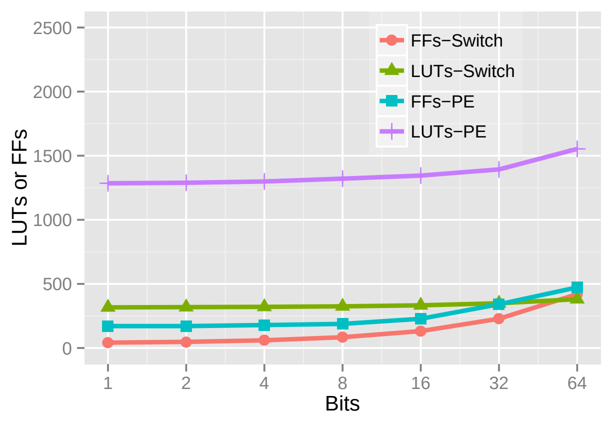
High-Level Synthesis. We note that the resource costs for the network are comparable to those reported in [9] adjusted for device generation.

The resource utilization of static hardware is shown in Figure 5 where both the datapath precision and schedule length are varied keeping the other quantity fixed. No addressing information needs to be routed in the switching network thereby lowering wiring requirements. The PE resource utilization rises rapidly with precision while the routing network costs stay relatively flat as they are dominated by a schedule length of 1024. When varying schedule length in Figure 5b, the FF utilization stays fixed as expected but the LUT counts increase to accommodate the rising schedule storage requirements. As future work, when BRAM usage rises above a certain threshold, we will consider offloading schedule storage to LUT RAMs instead, thereby keeping resouce utilization balanced.

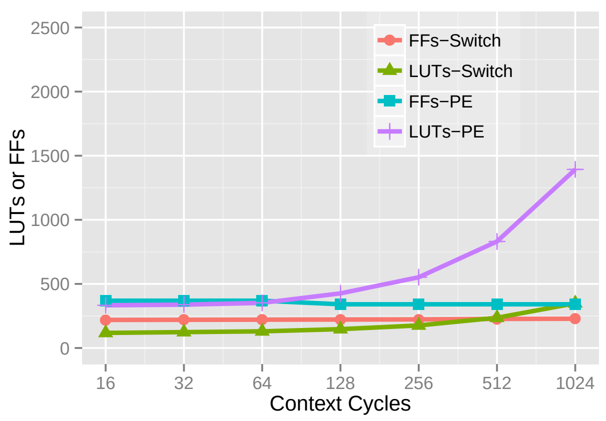
Finally in Figure 7, we show the ratio of LUT, FF and BRAM utilization for statically-scheduled and dynamically-scheduled architectures at 32b precision. We observe a ratio of 3–7× higher resource utilization for dynamic scheduling at small schedule lengths of 16 cycles. This is interesting but largely irrelevant as most dataflow graphs will require larger schedule lengths. For our experiments we report later in Section VI, we observe a peak requirement of ≈1K cycles. Hence, as we increase schedule length to 1024 cycles, resource costs of the static architecture increase causing a lowering of the utilization gap down to 2–3× for the various metrics. This is still a significant gap suggesting an ability to either (1) accommodate larger system sizes in the same physically-sized chip when using static scheduling, or (2) lower system costs by selecting a smaller, cheaper chip.

## IV. SPICE Circuit Simulator

In earlier work, we built an FPGA-based application accelerator for the SPICE circuit simulator using a hybrid architecture that combined VLIW, Token Dataflow and Streaming organizations [8], [6]. We resorted to using the dynamically-scheduled token dataflow organization for acceleration of Sparse LU factorization [7] because of our mistaken belief



(a) Varying Datapath Bits (Schedule Length=1024)



(b) Varying Schedule Length (Bits=32)

Fig. 5: FPGA Hardware Costs for Static Scheduling

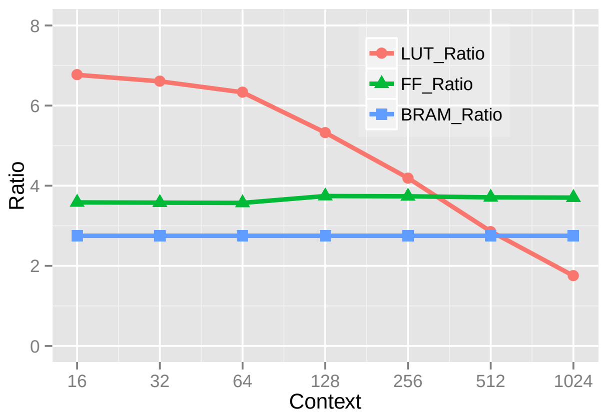


Fig. 7: Ratio of Resource Costs for Static and Dynamic Scheduling Hardware

that the graph sizes were large and challenging for static scheduling. In this paper, we rectify our mistake and overcome the scheduling challenge using a greedy XY router (as opposed to a traditional shortest-path router) to develop a new scheduler for dataflow acceleration of both **Model Evaluation** and **Matrix Solve** phase of SPICE. This enables us to develop a better understanding of performance limits and cost considerations when engineering application-specific dataflow machines. Thus we redesign a new SPICE accelerator that is built predominantly using dataflow techniques while relegating control-oriented tasks to the streaming subsystem.

## *A. SPICE Simulation Algorithm*

Analog SPICE simulations are time-consuming, hard-to-parallelize problems that represent a formidable performance challenge for conventional computing platforms. A typical SPICE simulation barely uses 1–10% of the CPU processing throughput.

SPICE simulates the dynamic analog behavior of a circuit described by non-linear differential equations. SPICE circuit equations model the linear (*e.g.* resistors) and non-linear (*e.g.* transistors) behavior of devices and the Kirchoff's Current Law at the different nodes and branches of the circuit. SPICE also captures the transient effects of devices such as inductors and capacitors. SPICE is an iterative non-linear differential equations solver that repeatedly computes small-signal linear operating-point approximations for the non-linear elements and discretizes continuous time behavior of time-varying elements. This system of equations is represented as a solution of $A\vec{x} = \vec{b}$, where $A$ is the matrix of circuit conductances, $\vec{b}$ is the vector of known currents and voltages in the circuit, and and $\vec{x}$ is the vector of unknown voltages and branch currents. The simulator updates $A$ and $\vec{b}$ from the device model equations that describe device transconductance (*e.g.*, Ohm's law for resistors, transistor I-V characteristics) in the **Model-Evaluation** phase. It then solves for $\vec{x}$ using a sparse linear matrix solver in the **Matrix-Solve** phase. Typical simulations require operating over millions of instantiations of irregular, floating-point dataflow graphs that represent transistor device equations as well as large-scale, million-entry sparse matrix representations of industrial circuits.

| **Characteristics** | **Model Evaluation** | **Matrix Solve** |
|---|---|---|
| Graph Complexity | ≈10-1K nodes/edges | ≈1K-1M nodes/edges |
| Graph Operations (ieee64) | multiply, add, sqrt, divide, exp, log | mult, add, divide |
| Reuse | *1K-1M iterations (benchmark-specific)* | |
| Reuse/Iteration | 1000s of times | Once per iteration |
| Graph Construction | Known at compile time | Known at start of runtime |
| Re-entrant | Yes | No |
| Static Schedulability | High | Low |

TABLE I: Characteristics of the Dataflow Graphs

## *B. Dataflow Characteristics*

The dataflow graphs extracted from the two SPICE phases exhibit unique characteristics as shown in Table I. These dataflow graphs are reused in a different manner motivating design transformations that make them better suited for dataflow

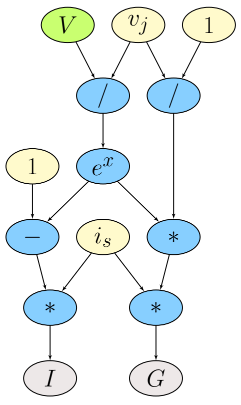
**Code Snippets** **Dataflow Graphs**

```
// foreach diode
for(int i=0;i<N;i++) {
    // current
    I[i] = is*(exp(V[i]/vj)-1);
    // conductance
    G[i] = (is/vj)*exp(V[i]/vj);
}
```

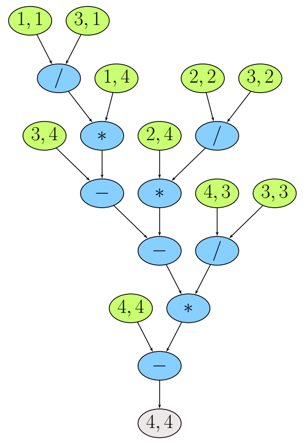
(a) Diode Code

```
// foreach column
for(int k=0;i<N;i++)
    for(int i=0;i<k;i++)
        for(int j=i;j<N;j++)
            if(nz)
                x[j] = x[j] -
                L[j][i]*x[i];
```

(d) LU Code



(b) Diode DFG



(c) LU DFG

Fig. 8: Representative Dataflow Graphs and their Input Program Descriptions

architectures. While it is possible to support dynamically-generated graphs on *dynamic dataflow* hardware, SPICE graphs are statically known at compile time or just at the start of runtime thereby satisfying *static dataflow* requirements.

**Model-Evaluation** dataflow graphs are small (10s–100s of nodes) and structurally known at compile-time since they are based on device physics equations that are well-known. However, they are invoked repeatedly (1K–1M times) in a loop depending on the size of the circuit being simulated. This re-entrant nature of the graph may motivate a Monsoon-like architecture with tagged tokens to isolate the individual invocation. However, we simplify this requirement by performing loop unrolling on the data-independent invocations and provisioning additional storage to hold the unrolled state. Additionally, since the device models require a variety of ALU operations such as multiply, add, divide, square-root, exponential and logarithm, we create specialized processors for each operator type and distribute instructions (dataflow nodes) accordingly.

**Matrix Solve** graphs capture structure of the circuit and are substantially larger (1K-1M nodes) depending on the size and connectivity of the circuit being simulated. These are not known until the start of the simulation, but once provided, the structure of the graph stays unchanged throughout the simulation. Ordinarily, LU factorization with full pivoting will not generate graphs that stay static in each iteration, but we perform static analysis on the matrix using the KLU solver to generate static pivot sequences that do not have any effect on convergence. This graph construction and scheduling cost can be a one-time overhead that is charged at the start of the iterative simulation and gets amortized across 1K–1M iterations as required by the circuit designer.

### C. *Opportunity for Static Scheduling*

While classic token dataflow organization offers a unique opportunity for parallelization of the SPICE-extracted dataflow graphs, we can also map these computations to statically-scheduled dataflow architectures. This is possible since the graphs are either known at compile time (for SPICE models across all executions) or the graphs stay structurally unchanged throughout the evaluation (for individual instances of SPICE execution), with only the propogated numerical values being changed in each invocation.

## V. Experimental Methodology

For our comparison study, we extract dataflow graphs for a set of SPICE benchmarks circuits and device models. We pick these benchmarks to cover a range of problem sizes. Dataflow graphs extracted from sparse matrices are much larger than those representing device equations and present a different stress scenario for the dataflow processors. We list the key properties of the dataflow graphs in Table II. We consider a range of dataflow graph sizes (including small ones) to faithfully identify trends.

To program our dataflow processor, we use the flow described in Figure 9. We develop a sparse graph pre-processor flow that constructs and distributes the dataflow graph for our benchmark dataflow graphs across parallel partitions. For placement of graph nodes on processors, we use MLPart to minimize bisection cut. The resulting data structures are copied over to the memory block in each processing element.

**Static Scheduler**: In the case of the statically-scheduled dataflow engines, we also generate a cycle-by-cycle programming context for the dataflow ROMs as well as the switching fabric. Our static router is a greedy XY router that searches for the earliest available scheduling cycle for a dataflow edge such

| Name | Nodes | Edges | Depth |
|---|---|---|---|
| Device Equations | | | |
| `diode_simple` | 43 | 58 | 14 |
| `vbic` | 225 | 250 | 34 |
| `jfet` | 91 | 120 | 13 |
| `hbt` | 612 | 669 | 32 |
| `bsim3v32_new` | 1459 | 2408 | 106 |
| Sparse Matrices | | | |
| `bomhof2_circuit_4465.mtx` | 20654 | 30094 | 131 |
| `bomhof2_circuit_4222.mtx` | 8018 | 11322 | 72 |
| `s208_401.mtx` | 94 | 106 | 19 |

TABLE II: Benchmark Properties

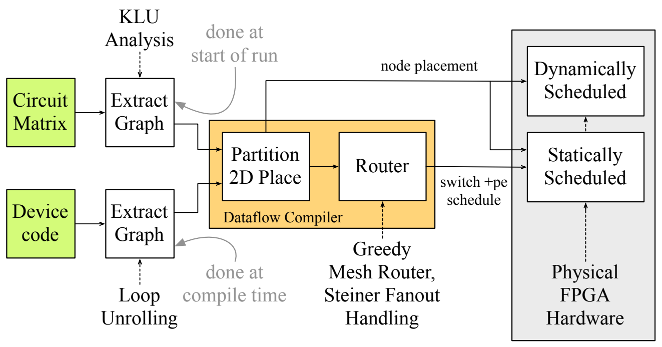


Fig. 9: Scheduler Flow

that dependency constraints are obeyed. Unlike [6], [9], this new static scheduler constrains the space dimension to pure XY routes (DOR) and explores freedom in the time dimension. While this is sub-optimal, it is a simple and fast router that outperforms dynamic scheduling by a healthy margin.

## VI. Experimental Evaluation

We are interested in identifying key performance indicators and scaling trends for the SPICE dataflow graphs. We vary the number of dataflow PEs (*i.e.* size of mesh) and measure the number of cycles required to route the entire workload. We report the results of our experiments by comparing cycle counts for Model Evaluation and Matrix Solve dataflow graphs extracted from SPICE.

**Matrix Solve** In Figure 10, we show results for the Matrix Solve benchmarks graphs. We observe that the dataflow architecture is not particularly suitable for small benchmarks like `s208_410.mtx` while the larger benchmarks show almost linear scaling in speedup even at large PE counts. Static scheduling offers as high as 4× improvement in cycles when compared to dynamic scheduling (Figure 12b) suggesting a large degree of available parallelism and high scheduling efficiency when routing the dependency dataflow graph.

**Model Evaluation** For Model Evaluation graphs, we observe fairly different scaling trends as shown in Figure 11. Since the device equation graphs are substantially smaller than the sparse matrix graphs, we note that performance scaling saturates beyond 16 PEs for almost all devices . In addition, static scheduling only offers $\approx 2\times$ speedup over dynamic scheduling for these benchmarks (Figure 12a). Overall speedups saturate around 16 PEs suggesting that a combination of dataflow organization and *tiling* at the optimal dataflow system size is needed to achieve scalable performance.

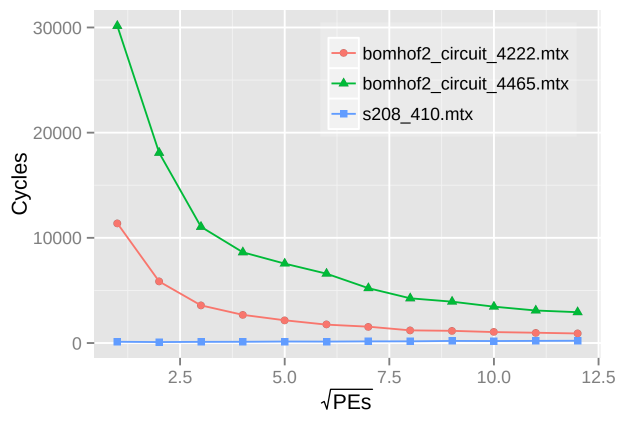
## VII. Conclusions and Future Work

Dataflow graph size and scheduling efficiency play a key role in determining the extent of gains achieved by statically-scheduled dataflow architectures over their dynamically-scheduled counterparts. For small dataflow graphs extracted from SPICE Model Evaluation we observe a $\approx 2\times$ performance benefit for the static implementation. For larger dataflow graphs generated from particular SPICE circuit matrices, we observe a much larger speedup of $\approx 4\times$ for the same comparison. When compounded with a 2–3× smaller hardware resource costs, we have a much larger compute density advantage with static scheduling. While the nature of these results should not be surprising, we identify trends and correlations that can predict this gap while also comparing RTL implementation costs.

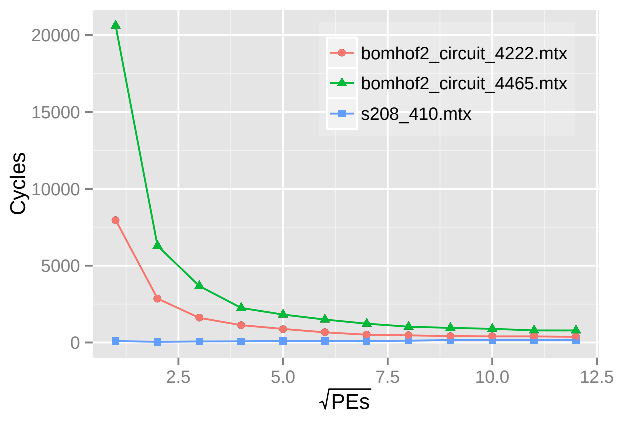
When scaling to even larger-scale dataflow problem sizes, we must prepare for challenges associated with: (1) dataflow graph construction times for exascale problems, (2) hardware scalability of triggering logic, (3) hybrid integration with classic ISA control processors, (4) scalable static schedulers.

## References

[1] D. Burger, S. W. Keckler, K. McKinley, and M. Dahlin. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 2004.
[2] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, 2000.
[3] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *ACM Comp. Architecture News*, 1975.
[4] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 1985.
[5] A. R. Hurson and K. M. Kavi. Dataflow Computers: Their History and Future. *Wiley Encyclopedia of Comp. Science and Engg.*, 2008.
[6] N. Kapre and A. DeHon. Accelerating SPICE Model-Evaluation using FPGAs. In *FCCM '09: Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, 2009.
[7] N. Kapre and A. DeHon. Parallelizing Sparse Matrix Solve for SPICE circuit simulation using FPGAs. In *Field-Programmable Tech.*, 2010.
[8] N. Kapre and A. DeHon. SPICE2: Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.
[9] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon. Packet switched vs. time multiplexed FPGA overlay networks. In *Proc. 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
[10] E. A. Lee. *Advanced Topics in Dataflow Computing*, chapter Static Scheduling of Data-Flow Programs for DSP. Prentice Hall, 1991.
[11] X. P. Ling and H. Amano. WASMII: a data driven computer on a virtual hardware. *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, 1993.
[12] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, 1990.
[13] A. Smith and J. Gray. Towards an Area-Efficient Implementation of a High ILP EDGE Soft Processor. In *FCCM '14: Proceedings of the 2014 22nd IEEE Symposium on Field Programmable Custom Computing Machines*, 2014.
[14] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

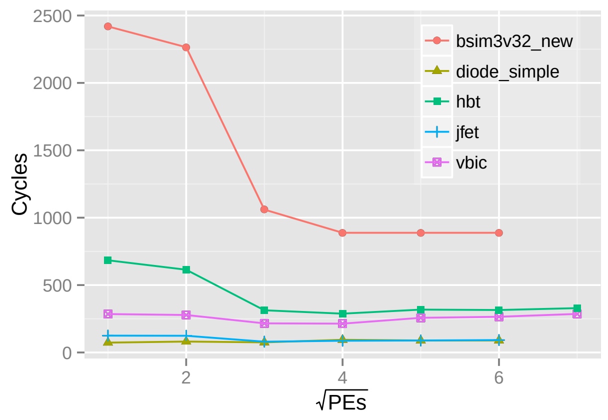


(a) Dynamic Scheduling Cycles

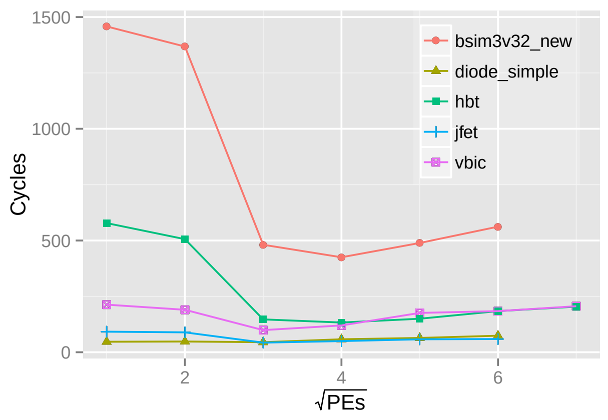


(b) Static Scheduling Cycles

Fig. 10: Performance Characterization of Matrix Solve Dataflow

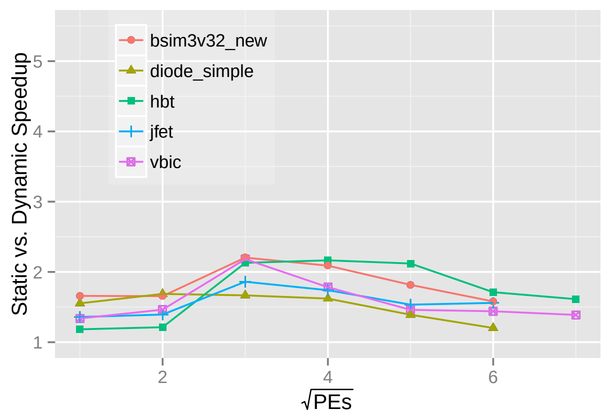


(a) Dynamic Scheduling Cycles

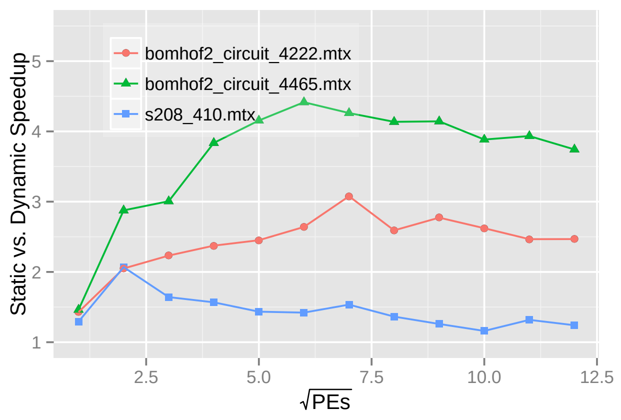


(b) Static Scheduling Cycles

Fig. 11: Performance Characterization of Model Evaluation Dataflow



(a) Model Evaluation



(b) Matrix Solve

Fig. 12: Comparing Performance of Static and Dynamic Scheduling