An NoC Traffic Compiler for efficient FPGA implementation of Sparse Graph-Oriented Workloads

Nachiket Kapre
Imperial College London
London, SW7 2AZ
nachiket@imperial.ac.uk

André DeHon
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

***Abstract*—Parallel graph-oriented applications expressed in the Bulk-Synchronous Parallel (BSP) and Token Dataflow compute models generate highly-structured communication workloads from messages propagating along graph edges. We can expose this structure to traffic compilers and optimization tools before runtime to reshape and reduce traffic for higher performance (or lower area, lower energy, lower cost). Such offline traffic optimization eliminates the need for complex, runtime NoC hardware and enables lightweight, scalable FPGA NoCs. In this paper, we perform load balancing, placement, fanout routing, and fine-grained synchronization to optimize our workloads for large networks up to 2025 parallel elements for BSP model, 25 parallel elements for Token Dataflow and 128 parallel elements for Static SIMD. This allows us to demonstrate speedups between 1.2× and 22× (3.5× mean), area reductions (number of Processing Elements) between 3× and 15× (9× mean) and dynamic energy savings between 2× and 3.5× (2.7× mean) over a range of real-world graph applications in the BSP compute model. We deliver speedups of 0.5-13× (geomean 3.6×) for Sparse Direct Matrix Solve (Token Dataflow compute model) applied to a range of sparse matrices when using a high-quality placement algorithm. We expect such traffic optimization tools and techniques to become an essential part of the NoC application-mapping flow.**

I. Introduction

Real-world communication workloads exhibit structure in the form of locality, sparsity, fanout distribution, and other properties. If this structure can be exposed to automation tools, we can reshape and optimize the workload to improve performance, lower area and reduce energy. In this paper, we develop a traffic compiler that exploits structural properties of Bulk-Synchronous Parallel communication workloads. This compiler provides insight into performance tuning of communication-intensive parallel applications. The performance and energy improvements made possible by the compiler allow us to build the NoC from simple hardware elements that consume less area and eliminate the need for using complex, area-hungry, adaptive hardware. We now introduce key structural properties exploited by our traffic compiler.

- When the natural communicating components of the traffic do not match the granularity of the NoC architecture, applications may end up being poorly load balanced. We discuss *Decomposition* and *Clustering* as techniques to improve load balance.

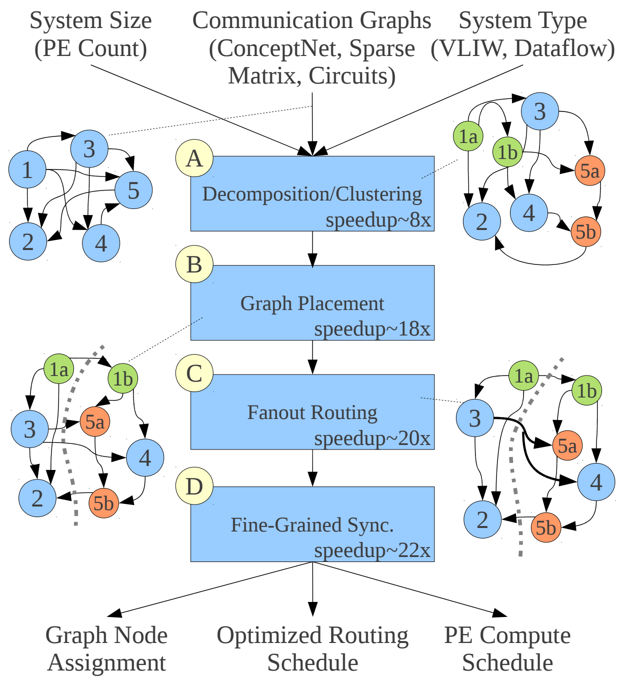


Fig. 1: NoC Traffic Compilation Flow
(annotated with `cnet-default` speedups at 2025 PEs)

- Most applications exhibit sparsity and locality; an object often interacts regularly with only a few other objects in its neighborhood. We exploit these properties by *Placing* communicating objects close to each other.
- Data updates from an object should often be seen by multiple neighbors, meaning the network must route the same message to multiple destinations. We consider *Fanout Routing* to avoid redundantly routing data.
- Applications that use barrier synchronization can minimize node idle time induced by global synchronization between the parallel regions of the program by using *Fine-Grained Synchronization*.

We show the compilation flow for the NoC in Figure 1 and illustrate the relative benefits using an example application graph. We also show representative speedups for the ConceptNet `cnet-default` workload after each step of the traffic compilation flow when compared to an unoptimized mapping.

The unoptimized graph has imbalanced, performance-limiting communication characteristics which get reshaped, reduced and redistributed for an efficient execution by the traffic compiler. While these optimizations have been discussed independently in the literature extensively (*e.g.* [1]–[5]), we develop a toolflow that auto-tunes the control parameters of these optimizations per workload for maximum benefit and provide a quantification of the cumulative benefit of applying these optimizations to various applications in fine-grained, onchip network settings. This quantification further illustrates how the performance impact of each optimization changes with NoC size. The key contributions of this paper include:

- Development of a traffic compiler for fine-grained applications described using the BSP, Token Dataflow and Static SIMD compute models.
- Use of communication workloads extracted from ConceptNet (BSP), Sparse Matrix-Vector Multiply (BSP), Bellman-Ford (BSP), and Sparse Direct Matrix Solve (Token Dataflow) running on range of real-world circuits and graphs.
- Quantification of cumulative benefits of each stage of the compilation flow (performance, area, energy) compared to the unoptimized case.

II. Background

A. Applications and Compute-Models

We consider two compute models and associated applications: (1) Graphstep [6], and (2) Token Dataflow [7].

Graphstep: Parallel graph algorithms are well-suited for concurrent processing on FPGAs. We describe graph algorithms in a Bulk-Synchronous Parallel (BSP) compute model [8] and develop an FPGA system architecture [6] for accelerating such algorithms. The compute model defines the intended semantics of the algorithm so we know which optimizations preserve the desired meaning while reducing NoC traffic. The graph algorithms are a sequence of steps where each step is separated by a global barrier. In each step, we perform parallel, concurrent operations on nodes of a graph data-structure where all nodes send messages to their neighbors while also receiving messages. The graphs in these algorithms are known when the algorithm starts and do not change during the algorithm. Our communication workload consists of routing a set of messages between graph nodes. We route the same set of messages, corresponding to the graph edges, in each epoch.

Token Dataflow: Lightweight-processing of sparse dataflow graphs can be efficiently accelerated using FPGAs. In [7] we show how to accelerate Sparse Matrix-Solve computation generated by the KLU [9] solver using this approach. The dataflow compute model enables parallel distributed operations in the sparse dataflow graph by implementing the *dataflow firing rule*. According to this rule each node in the graph will *fire* when it receives all its inputs. Instead of a global barrier each node implements its own local synchronization operation. Each token is a message that is routed along an edge of the dataflow graph. We evaluate the complete graph by propagating tokens from the graph inputs all the way to the outputs in dataflow fashion. Our approach is similar to the implementation in [10]. Token propagation along parallel paths allow this token-passing architecture to exploit parallelism in the graph. Our workload is the ordered set of messages that are activated and routed according to the dependencies in the graph.

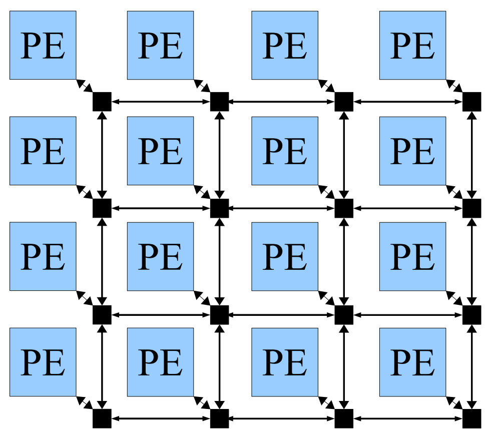
Applications in the compute models we consider generate traffic with a variety of communication characteristics (*e.g.* locality, sparsity, multicast) which also occur in other applications and compute models as well. Our traffic compiler exploits the *a priori* knowledge of structure-rich communication workloads (see Section IV-A) to provide performance benefits. Our approach differs from some recent NoC studies that use statistical traffic models (*e.g.* [11]–[14]) and random workloads (*e.g.* [15]–[17]) for analysis and experiments. Statistical and random workloads may exaggerate traffic requirements and ignore application structure leading to over-provisioned NoC resources and missed opportunities for workload optimization. We use real workloads generated from three different compute models to demonstrate the value and generality of our parallel approach.

Other Studies: In [11], the authors demonstrate a 60% area reduction along with an 18% performance improvement for well-behaved workloads. In [13], the authors show a 20% reduction in buffer sizes and a 20% frequency reduction for an MPEG-2 workload. In [15], the authors deliver a 23% reduction in time, a 23% reduction in area as well as a 38% reduction in energy for their design. We demonstrate better performance (95% reduction in runtime), lower area requirements (90% area savings) and lower energy consumption (90% less energy). Our approach is designed to deliver the larger, order-of-magnitude improvement because our systems (1) route fine-grained message-passing workloads (2) utilize a high-throughput design of the Processing Elements, and (3) support scalability to larger system sizes. In contrast, the other studies attempt to optimize NoCs running restrictive, coarse-grained application workloads which reduce the impact of traffic characteristics on overall system behavior.

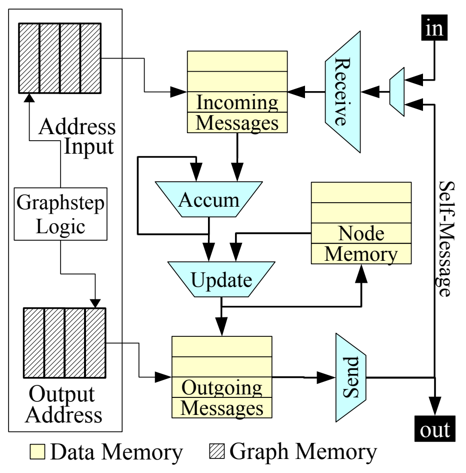
B. Architecture

We organize our FPGA NoC as a bidirectional 2D-mesh [18] with a packet-switched routing network as shown in Figure 2(a). The application graph is distributed across the Processing Elements (PEs) which are specialized to process graph nodes for the different compute models. Portions of the application graph are stored in local on-chip memories in each PE. The PE is internally pipelined and capable of injecting and receiving a new packet in each cycle. The PE can simultaneously handle incoming and outgoing messages.

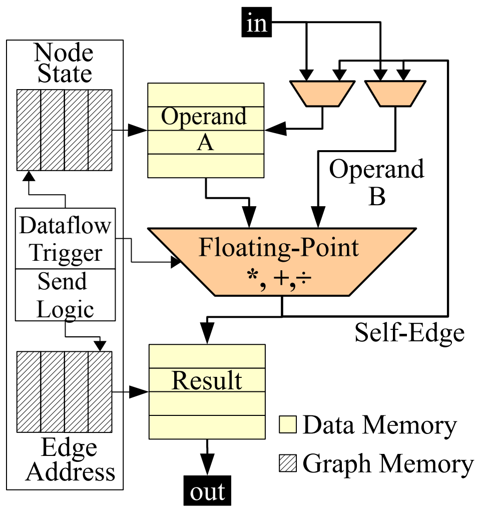
BSP PE: For the BSP PE shown in Figure 2(b), each PE performs accumulate and update computations on each node as defined by the BSP graph algorithm. The internal PE pipelines are managed by the GraphStep Logic controller. During execution, the controller iterates through all local nodes and generates outbound traffic that is routed over the packet-



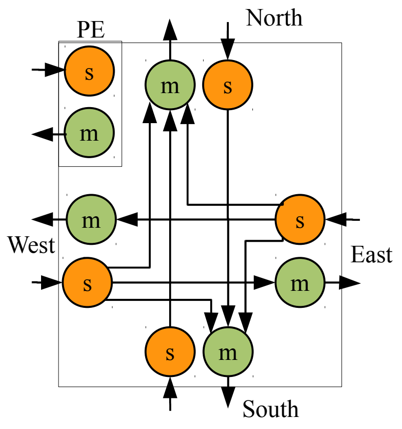
(a) Architecture of the NoC



(b) Graphstep PE



(c) Dataflow PE



(d) Mesh Switch (DOR, Implicit PE connections)

Fig. 2: NoC Architecture and Organization

switched network. Inbound traffic is stored in the incoming message buffers of each PE. Once all messages have been received, a barrier is detected using a global reduce tree (a bit-level AND-reduce tree). The graph application proceeds through multiple global barriers until the algorithm terminates.

Token Dataflow PE: For the Token Dataflow PE show in Figure 2(c), each PE implements the *dataflow firing rule* by keeping track of the number of messages received on each graph node. In each execution of the graph, the PE starts processing the graph from the inputs nodes and successively propagates the computation through the levels of the graph to the outputs. We do not use global barriers and instead allow distributed local barriers at each node. After a node is fired, it is inserted into the result queue that injects message into the network for each recipient of the node. This propagates the computation through the levels of the graph. We declare termination when all network traffic and PE activity has quiesced using a global reduce tree.

Network Switch: Each switch in the bidirectional 2D mesh supports fully-pipelined operation using composable *Split* and *Merge* units as shown in Figure 2(d). The switches in the bidirectional mesh network implement the Dimension-Ordered Routing (DOR) algorithm [19] that is simplest to realize in hardware and widely used in NoC designs. We discuss additional implementation parameters in Section IV-B. Prior to execution, the traffic compiler is responsible for allocating graph nodes to PEs.

We measure network performance as the number of clock cycles ($Barrier\_Cycles$) required for one epoch between barriers, including both computation and all messages routing. We report speedups as $Speedup = \frac{Barrier\_Cycles_{unoptimized}}{Barrier\_Cycles_{optimized}}$

III. Optimizations

In this section, we describe a set of optimizations performed by our traffic compiler. The compiler accepts the graph structure from the application and maps it to the NoC architecture. It suitably modifies the graph structure (replacing nodes and edges) and generates an assignment of graph nodes to the PEs of the NoC. The traffic compiler also selects the type of synchronization implemented in the PEs. It is a fully automated flow that sequences the different graph optimizations to generate an optimized mapping.

A. Decomposition

Ideally for a given application, as the PE count increases, each PE holds smaller and smaller portions of the workload. For graph-oriented workloads, unusually large nodes with a large number of edges (*i.e.* nodes that send and receive many messages) can prevent the smooth distribution of the workload across the PEs. As a result, performance is limited by the time spent sending and receiving messages at the largest node (streamlined message processing in the PEs implies work $\propto$ number of messages per node). For example, the ConceptNet `cnet-default` workload has a communicating node with 16K input edges and 36K outgoing edges (about 10% of total edges, See Table I). For a fully-streamlined, high-throughput PE operation (1 cycle/network operation) we will need to spend at least 16K cycles receiving messages and 36K cycles sending messages irrespective of the number of PEs in the parallel NoC. *Decomposition* is a strategy where we break down large nodes into smaller nodes (either inputs, outputs or both can be decomposed) and distribute the work of sending and receiving messages at the large node over multiple PEs. The idea is similar to that used in synthesis and technology mapping of logic circuits [1]. Fig. 3 illustrates the effect of decomposing a node. Node 5 with 3 inputs gets *fanin-decomposed* into Node 5a and 5b with 2 inputs each thereby reducing the serialization at the node from 3 cycles to 2. Similarly, Node 1 with 4 outputs is *fanout-decomposed* into Node 1a and 1b with 3 outputs and 2 outputs each. Greater benefits can be achieved with higher-fanin/fanout nodes as is the case with `cnet-default` workload (see Table I).

In general, when the output from the graph node is a result which must be multicast to multiple outputs, we can easily build an output fanout tree to decompose output routing. However, input edges to a graph node can only be decomposed when the operation combining inputs is associative. ConceptNet and Bellman-Ford (discussed later in Section IV-A) permit input decomposition since nodes perform simple integer *sum* and *max* operations which are associative and can be decomposed. However, Matrix Multiply nodes perform non-associative *floating-point accumulation* over incoming values which cannot be broken up and distributed.

We implement the decomposition phase by constructing an n-ary tree that replaces the high-fanin or high-fanout node. As as example, consider node 6 in Figure 4 with 5 inputs. We construct a suitable 2-ary tree rooted at node 6 and assign the fanin nodes to the leaves of this tree. When the number

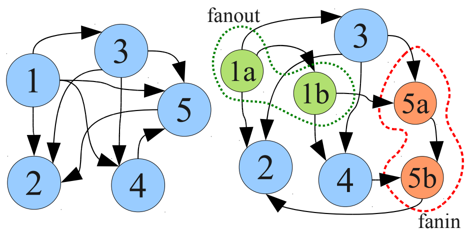


Fig. 3: *Decomposition*

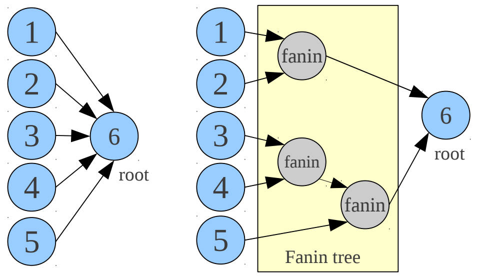


Fig. 4: *Fanin Decomposition Tree Example*

of inputs are not a power-of-2, we generate an appropriate imbalanced tree to accommodate all inputs. Presently, our algorithm orders the inputs or outputs to the leaves arbitrarily while showing performance and scalability improvements. In the future, we can also consider sophisticated tree construction algorithms that attempt to carefully minimize the critical path or post-placement locality during leaf assignment.

B. Clustering

While *Decomposition* is necessary to break up large nodes, we may still have an imbalanced system if we randomly place nodes on PEs. Random placement fails to account for the varying amount of work performed per node. Lightweight *Clustering* is a common technique used to quickly distribute nodes over PEs to achieve better load balance (*e.g.* [2]). We use a greedy, linear-time *Clustering* algorithm similar to the *Cluster Growth* technique from [2]. We start by creating as many "clusters" as PEs and randomly assign a seed node to each cluster. We then pick nodes from the graph and greedily assign them to the PE that least increases cost. The cost function ("Closeness metric" in [2]) is chosen to capture the amount of work done in each PE including sending messages, receiving messages or computing on a node. This is expressed as shown in Figure 5. In the equation, we compute total cost as a sum of communication and computation costs. The clustering algorithm then picks the cluster which has the smallest $Cost$. Communication cost is simply the larger of the send and receive cycles. Compute cost is a function of the number of cycles in the pipelined compute datapath (for non-associative computation we must add the total pipeline datapath cycles per edge, $Cycles$). We can explore other greedy clustering packages (*e.g.* T-Vpack [20]) as part of future enhancements to the compiler.

$$\begin{aligned}
Inputs &= \sum_{node \in cluster} Input\_Edges(node) && (1)\\
Outputs &= \sum_{node \in cluster} Output\_Edges(node) && (2)\\
MaxIn &= \max_{node \in cluster} Input\_Edges(node) && (3)\\
NetworkIO &= \max(Inputs, Outputs) && (4)\\
Compute &= \max(Inputs, MaxIn * Cycles) && (5)\\
Cost &= NetworkIO + Compute && (6)
\end{aligned}$$

Fig. 5: Clustering Cost Function

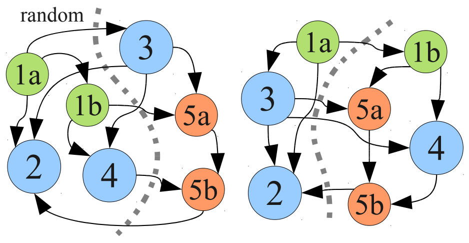


Fig. 6: *Placement*
(Random Placement vs. Good Placement)

C. Placement

Object communication typically exhibits locality. A random placement ignores this locality resulting in more traffic on the network. Consequently, random placement imposes a greater traffic requirement which can lead to poor performance, higher energy consumption and inefficient use of network resources. We can *Place* communicating nodes close to each other to minimize traffic requirements and get better performance than random placement. The benefit of performing placement for NoCs has been discussed in [3]. Good placement reduces both the number of messages that must be routed on the network and the distance which each message must travel. This decreases competition for network bandwidth and lowers the average latency required by the messages. Fig. 6 shows a simple example of good *Placement*. A random partitioning of the application graph may bisect the graph with a cut size of 6 edges (*i.e.* 6 messages must cross the chip bisection). Instead, a high-quality partitioning of the graph will find a lower cut width size of 4. The load on the network will be reduced since 2 fewer messages must cross the bisection. In general, *Placement* is an NP-complete problem, and finding an optimal solution is computationally intensive. We use a fast multi-level partitioning heuristic MLPART [21] that iteratively clusters nodes and moves the clustered nodes around partitions to search for a better quality solution. In future, we can improve placement quality with a slower simulated-annealing heuristic.

D. Fanout Routing

Some applications may require multicast messages (*i.e.* single source, multiple destinations). Our application graphs contain nodes that send the exact same message to their destinations. Routing redundant messages is a waste of network resources. We can use the network more efficiently with *Fanout Routing* which avoids routing redundant messages. This has been studied extensively by Duato *et al.* [4]. If many destination nodes reside in the same physical PE, it is possible to send only one message instead of many, duplicate messages to the PE. For this to be beneficial, there needs to be at least two sink nodes in some destination PE. The PE will then internally distribute the message to the intended recipients. This is shown in Fig. 7. The fanout edge from Node 3 to Node 5a and Node 4 can be replaced with a shared edge as shown. This reduces the number of messages crossing the bisection by 1. This optimization works best at reducing traffic and message-injection costs at low PE counts. As PE counts increase we have more possible destinations for the outputs and fewer shareable nodes in the PEs resulting in decreasing benefits. We realize this optimization with no hardware overheads. We simply configure the data-structures with appropriate addresses for indexing into the shared message memory.

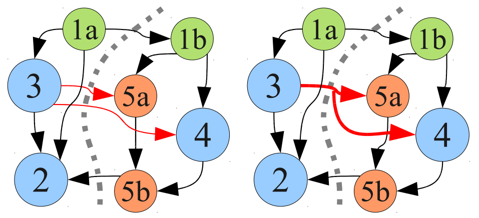


Fig. 7: *Fanout-Routing*

E. Fine-Grained Synchronization

In parallel programs with multiple threads, synchronization between the threads is sometimes implemented with a global barrier for simplicity. However, the global barrier may artificially serialize computation. Alternately, the global barrier can be replaced with local synchronization conditions that avoid unnecessary sequentialization. Techniques for eliminating such barriers have been previously studied [5], [22]. In the BSP compute model discussed in Section II, execution is organized as a series of parallel operations separated by barriers. We use one barrier to signify the end of the communicate phase and another to signify the end of the compute phase. If it is known prior to execution that the entire graph will be processed, the first barrier can be eliminated by using local synchronization operations. A node can be permitted to start the compute phase as soon as it receives all its incoming messages without waiting for the rest of the nodes to have received their messages. This prevents performance from being limited by the sum of worst-case compute and communicate latencies when they are not necessarily coupled. This is implemented by adding extra logic and state to store and update messages received per node as well as the total message count per node. We show the potential benefit of *Fine-Grained Synchronization* in Fig. 8. Node 2 and Node 3 can start their *Compute* phases after they have received all their inputs messages. They do not need to wait for all other nodes to receive all their messages. This optimization enables the *Communicate* phase and the *Compute* phase to be overlapped.

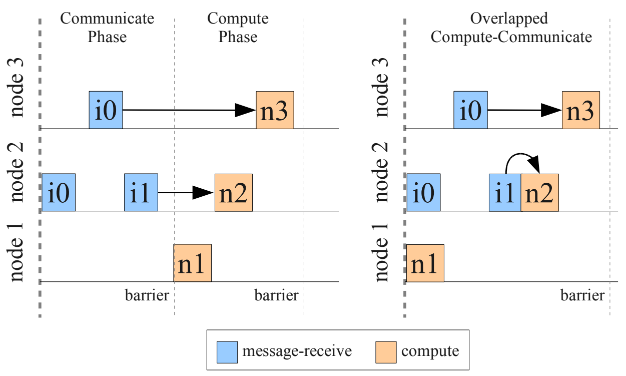


Fig. 8: *Fine-Grained Synchronization*

IV. Experimental Setup

A. Workloads

We generate workloads from a range of applications mapped to the BSP compute model and the Token Dataflow model. We choose applications that cover different domains including AI, Scientific Computing and CAD optimization that exhibit important structural properties.

1) ConceptNet (BSP): ConceptNet [23] is a common-sense reasoning knowledge base described as a graph, where nodes represent concepts and edges represent semantic relationships. Queries to this knowledge base start a *spreading-activation* algorithm from an initial set of nodes. The computation spreads over larger portions of the graph through a sequence of steps by passing messages from activated nodes to their neighbors. In the case of complex queries or multiple simultaneous queries, the entire graph may become activated after a small number of steps. We route all the edges in the graph representing this worst-case step. In [6], we show a per-FPGA speedup of 20× when comparing a Xilinx Virtex-2V6000 FPGA to an optimized sequential implementation on a 3.4 GHz Pentium-4 Xeon . We use the BSP compute model to capture the semantics of the computation and implement the application on a Graphstep PE (Figure 2(b)) with an associative datapath (input edges can be summarized associatively as they arrive without being sequentialized with a particular ordering).

2) Sparse Matrix-Vector Multiply (BSP): Iterative Sparse Matrix-Vector Multiply (SMVM) is the dominant computational kernel in several numerical routines (*e.g.* Conjugate Gradient, GMRES). In each iteration a set of dot products between the vector and matrix rows is performed to calculate new values for the vector to be used in the next iteration. We can represent this computation as a graph where nodes represent matrix rows and edges represent the communication of the new vector values. The graph captures the sparse communication structure inherent in the dot-product expression. In each iteration messages must be sent along all edges; these edges are multicast as each vector entry must be sent to each row graph node with a non-zero coefficient associated with the vector position. We use sample matrices from the Matrix Market benchmark [24]. Our traffic optimized implementation in this paper improves over the ≈2× speedups demonstrated previously in [25] when comparing a Xilinx Virtex-2V6000 with an Itanium 2 processor. We use the BSP compute model to express the sparse matrix computation and realize the processing on a Graphstep PE (Figure 2(b)) with a non-associative datapath.

3) Bellman-Ford (BSP): The Bellman-Ford algorithm solves the single-source shortest-path problem, identifying any negative edge weight cycles, if they exist. It finds application in CAD optimizations like Retiming, Static Timing Analysis and FPGA Routing where the graph structure is a representation of the physical circuit. Nodes represent gates in the circuit while edges represent wires between the gates. The algorithm simply relaxes all edges in each step until quiescence. A relaxation consists of computing the minimum at each node over all weighted incoming message values. Each node then communicates the result of the minimum to all its neighbors to prepare for the next relaxation. Again, we capture this computation in the BSP compute model and implement it on a Graphstep PE (Figure 2(b)) with an associative datapath.

4) Sparse Direct Matrix Solve in SPICE (Token Dataflow): Matrix Solve computation on sparse matrices is a key repetitive component of many applications like the SPICE circuit simulator. For SPICE, we prefer to use sparse direct solver techniques than SMVM-based (see Section IV-A2) iterative techniques for reasons of robustness at the expense of parallelism. We integrate the KLU solver with SPICE to expose parallelism in the computation with a one-time symbolic analysis at the start of the simulation. We extract sparse dataflow graphs for the LU factorization and Front-Solve and Back-Solve phases which we distribute across our parallel NoC architecture. Each node in the graph represents a floating-point operation (add, multiply, or divide) while each edge represents a dependency in the calculation. The original non-zero values in the sparse matrix represent graph inputs and the factored L and U non-zero values represent the graph outputs. We evaluate the graph by propagating tokens from the inputs of the graph to its outputs. This computation is processed on the NoC architecture using a Dataflow PE (Figure 2(c)). In [26], we show how to construct the dataflow graph and accelerate the Sparse Matrix Solve computation by 0.6–7.1× when comparing a Xilinx Virtex-6 LX760 with an optimized sequential implementation on an Intel Core i7 965.

B. NoC Timing and Power Model

All our experiments use a single-lane, bidirectional-mesh topology that implements a Dimension-Ordered Routing function. The network for Matrix-Vector Multiply and Sparse Direct Matrix Solve experiments is 84-bits wide (64-bits double-precision data, 20-bits header/address) while the network for ConceptNet and Bellman-Ford experiments is 52-bits wide (32-bits integer data, 20-bits header/address). The switch is internally pipelined to accept a new packet on each cycle (see Figure 2(a)). Different routing paths take different latencies inside the switch (see Table II). We pipeline the wires between the switches for high performance (counted in terms of cycles required as T_{wire}). The PEs are also pipelined

TABLE I: Application Graphs

Graph	Nodes	Edges	Max Fanin	Max Fanout
BSP Compute Model [6]				
ConceptNet				
`cnet-small`	14556	27275	226	2538
`cnet-default`	224876	553837	16176	36562
Matrix-Multiply				
`add20`	2395	17319	124	124
`bcsstk11`	1473	17857	27	30
`fidap035`	19716	218308	18	18
`fidapm37`	9152	765944	255	255
`gemat11`	4929	33185	27	28
`memplus`	17758	126150	574	574
`rdb32001`	3200	18880	6	6
`utm5940`	5940	83842	30	20
Bellman-Ford				
`ibm01`	12752	36455	33	93
`ibm05`	29347	97862	9	109
`ibm10`	69429	222371	137	170
`ibm15`	161570	529215	267	196
`ibm16`	183484	588775	163	257
`ibm18`	210613	617777	85	209
Token Dataflow Compute Model [7]				
`bcspwr01`	753	985	3	6
`mux8`	1037	1395	3	8
`ringosc`	2883	3866	3	4
`psadmit1`	9814	13356	3	10
`dac`	43000	67265	3	10
`psadmit2`	22259	30108	3	11
`sandia01`	40400	55765	3	8
`sandia02`	40400	55765	3	8
`s208`	43055	62067	3	11
`bcspwr09`	221807	391654	3	53
`s298`	70928	106247	3	13
`s344`	70666	103314	3	12
`s349`	73914	108888	3	14
`s382`	81060	119475	3	16
`s444`	90288	133901	3	16
`s386`	100637	151868	3	20
`s510`	220092	380930	3	54
`s526`	146442	228017	3	26
`s641`	212474	348453	3	39
`10stages`	124720	178396	3	8
`circuit2`	416454	747587	3	172

to start processing a new edge every cycle. ConceptNet and Bellman-Ford compute simple sum and max operations while Matrix-Vector Multiply performs floating-point $accumulation$ on the incoming messages. The Sparse Direct Matrix Solve algorithm performs floating-point $accumulation$ and $divide$ operations on the incoming messages as appropriate. For our floating-point benchmarks, we consider double-precision implementations of all operations. Each computation on the edge then takes 1–57 cycles of latency to complete (see Table II). We estimate dynamic power consumption in the switches using XPower [27]. Dynamic power consumption at different switching activity factors is shown in Table III. We extract switching activity factor in each Split and Merge unit from our packet-switched simulator. When comparing dynamic energy, we multiply dynamic power with simulated cycles to get energy. We generate bitstreams for the switch and PE on a Xilinx Virtex-5 LX110T FPGA [27] to derive our timing and power models shown in Table II and Table III.

TABLE II: NoC Timing Model

Mesh Switch	Latency
$T_{through}$ (X-X, Y-Y)	2
T_{turn} (X-Y, X-Y)	4
$T_{inteface}$ (PE-NoC, NoC-PE)	6
T_{wire} (GraphStep NoC)	2
T_{wire} (Token Dataflow NoC)	5
Processing Element	Latency
T_{send}	1
$T_{receive}$ (ConceptNet, Bellman-Ford)	1
$T_{receive}$ (Matrix-Vector Multiply)	9
T_{add} (Sparse Matrix Solve)	8
$T_{multiply}$ (Sparse Matrix Solve)	10
T_{divide} (Sparse Matrix Solve)	57

C. Packet-Switched Simulator

We use a Java-based cycle-accurate simulator that implements the timing model described in Section IV-B for our evaluation. The simulator models both computation and communication delays, simultaneously routing messages on the NoC and performing computation in the PEs. Our results in Section V report performance observed on cycle-accurate simulations of different circuits and graphs. The application graph is first transformed by a, possibly empty, set of optimizations from Section III before being presented to the simulator.

V. Evaluation

We now examine the impact of the different optimizations on various workloads to quantify the cumulative benefit of our traffic compiler. Our performance baseline is an unoptimized, unprocessed, barrier-synchronized graph workload which is randomly distributed across the NoC PEs. We order the optimization appropriately to analyze their additive impacts. We first show relative scaling trends for the total routing time for the `bcsstk11` benchmark to identify potential performance bottlenecks. We then quantify the impact of each optimization in systematically eliminating these bottlenecks. Initially, we load balance our workloads by performing *Decomposition*. We then determine how the workload gets distributed across PEs using *Clustering* or *Placement*. Finally, we perform *Fanout Routing* and *Fine-Grained Synchronization* optimizations. We illustrate scaling trends of individual optimizations using a single illustrative workload for greater clarity. At the end, we show cumulative data for all benchmarks together.

A. Performance Scaling Trends

Ideally, as PE counts increase, the application performance scales accordingly; ($T_{parallel} = T_{sequential}/PEs$) where $T_{sequential}$ = sequential time. However, applications rarely exhibit such ideal behavior. We recognize three potential bottlenecks that can prevent ideal scaling which we illustrate in Figure 9 for the `bcsstk11` benchmark.

1) Serialization: This metric measures the number of cycles spent injecting or receiving messages at the NoC-PE interface. We measure this as follows:

$$T_{input} = \quad (N_{input} + N_{self}) \times T_{send} \qquad (7)$$

TABLE III: NoC Dynamic Power Model

Datawidth (Application)	**Block**	Dynamic Power at diff. activity (mW) 0%	25%	50%	75%	100%
52 (ConceptNet, Bellman-Ford)	Split	0.26	1.07	1.45	1.65	1.84
	Merge	0.72	1.58	2.1	2.49	2.82
84 (Matrix-Vector Multiply, Sparse Marix Solve)	Split	0.32	1.35	1.78	2.02	2.26
	Merge	0.9	1.87	2.45	2.88	3.25

$$T_{output} = \quad (N_{output} + N_{self}) \times T_{receive} \tag{8}$$

In Equation 7 and Equation 8, N_{input} = number of messages entering the PE, N_{output} = number of messages leaving the PE, N_{self} = number of self-messages in the PE, T_{send} and $T_{receive}$ = number of cycles between successive sends and receives respectively. We engineer our PEs to handle an external input, output or self message in one cycle (T_{send}=$T_{receive}$=1). The internal memory architecture of our PE requires N_{self} to be counted on both ports. Since our input and output interfaces work independently and simultaneously, we define the total serialization cost as follows:

$$T_{serialization} = \quad \max(T_{input}, T_{output}) \tag{9}$$

We expect that, for ideal scaling, the number of serialization cycles decrease with increasing PE counts. We distribute both the computation and the communication over more PEs. However, communication from very large graph nodes (*i.e.* large number of edges) will cause a serial bottleneck at the PE-NoC interface. In Section III we discussed *Decomposition* (Section III-A), *Clustering* (Section III-B) and *Fanout Routing* (Section III-D) as strategies to avoid this bottleneck.

2) Bisection: This metric measures the number of cycles required for messages to cross the chip bisection. If the volume of NoC traffic crossing the chip bisection is larger than the number of physical wires (NoC channels in the bisection × Channel Width) then the bisection must be reused.

$$T_{cut} = \quad \left\lceil \frac{N_{messages}}{N_{wires}} \right\rceil \tag{10}$$

The top-level bisection may not be the largest bottleneck in the network. Hence, we consider several hierarchical cuts (horizontal and vertical cuts for a mesh topology) and identify the most limiting of cuts ($T_{cut i}$):

$$T_{bisection} = \max_{\text{cuts } i} (T_{cut i}) \tag{11}$$

For applications with high locality, the amount of traffic crossing the bisection is low (when placed properly) and bandwidth does not become a bottleneck. Conversely, for application with low locality, a larger number of messages need to cross the bisection and bisection bandwidth can become a bottleneck. In the Section III, we considered *Placement* (Section III-C) and to a lesser extent *Clustering* (Section III-B) to reduce bisection bandwidth requirements.

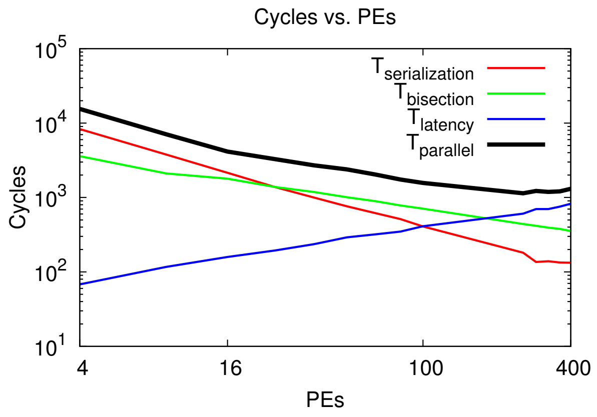


Fig. 9: Explaining performance scaling of total cycles for `bcsstk11` benchmark

3) Latency: This metric measures the sum of switch latencies and wire latencies along the worst-case message path in the NoC assuming no congestion.

$$T_{path} = \sum_{switches\ j} T_{switch_j} + \sum_{wires\ k} T_{wire_k} \tag{12}$$

$$T_{latency} = \max_{\text{messages } i} (T_{path_i}) \tag{13}$$

For barrier-synchronized workloads, all data is routed from sources to sinks in an epoch. At small PE counts, the number of cycles required to cross the network will be small compared to serialization or bisection cycles. However as the PE counts increase, the latency in the network will also increase and eventually dominate both serialization and bisection. In the high latency regime, latency hiding techniques like *Fine-Grained Synchronization* (Section III-E) that overlap compute and communicate phases become necessary.

In Figure 9, we observe that at low PE counts (PE<25), most cycles are dedicated towards serialized processing at the NoC-PE interfaces. As we increase the number of PEs (25<PE<128), the number of messages in the network increases causing the network to become bandwidth bottlenecked. Eventually, at high PE counts (PE>128), performance is dominated by latency.

B. Impact of Individual Optimizations

1) Decomposition: In Fig. 10, we show how the ConceptNet `cnet-default` workload scales with increasing PE counts under *Decomposition*. We observe that, *Decomposition* allows the application to continue to scale up to 2025 PEs and possibly beyond. Without *Decomposition*, performance quickly runs into a serialization bottleneck due to large nodes

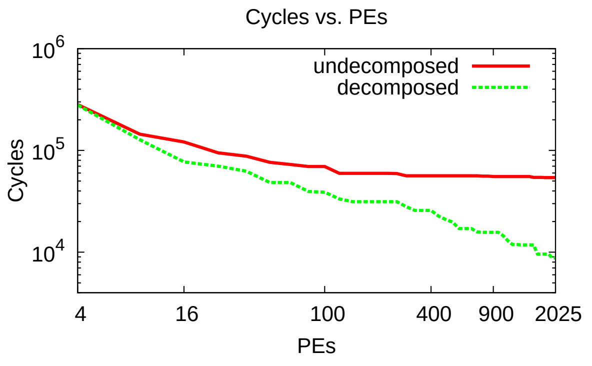


Fig. 10: *Decomposition* (`cnet-default`)

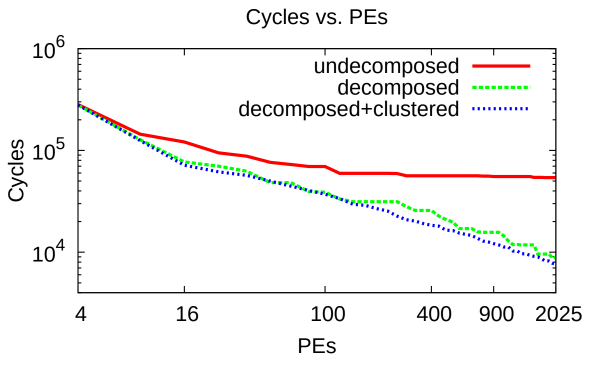


Fig. 12: *Decomposition* and *Clustering* (`cnet-default`)

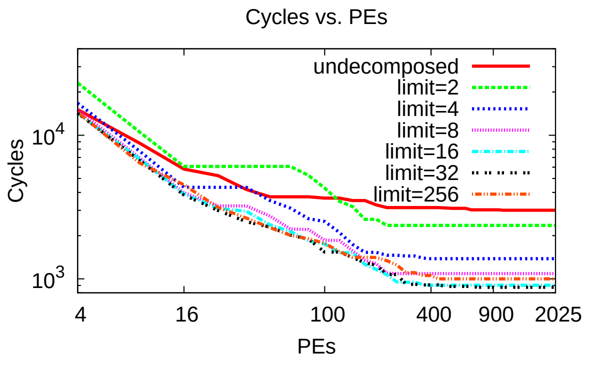


Fig. 11: *Decomposition Limits* (`cnet-small`)

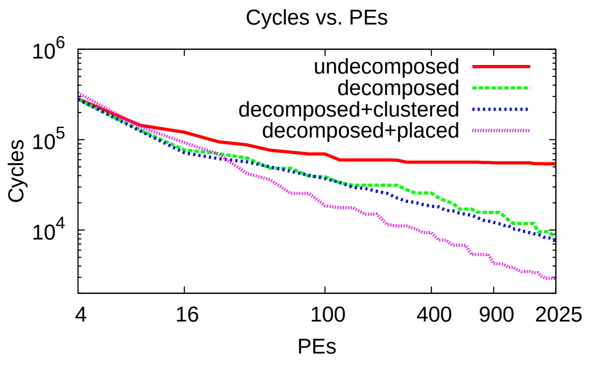


Fig. 13: *Decomposition*, *Clustering* and *Placement* (`cnet-default`)

as early as 100 PEs. The decomposed NoC workload manages to outperform the undecomposed case by 6.8× in performance. However, the benefit is lower at low PE counts, since the maximum logical node size becomes small compared to the average work per PE. Additionally, decomposition is only useful for graphs with high degree (see Table I). In Figure 11 we show how the *decomposition limit* control parameter impacts the scaling of the workload. As expected, without decomposition, performance of the workload saturates beyond 32 PEs. Decomposition with a limit of 16 or 32 allows the workload to scale up to 400 PEs and provides a speedup of 3.2× at these system sizes. However, if we attempt an aggressive decomposition with a limit of 2 (all decomposed nodes allowed to have a fanin and fanout of 2) performance is actually worse than undecomposed case between 16 and 100 PEs and barely better at larger system sizes. At such small decomposition limits, performance gets worse due to an excessive increase in the workload size (*i.e.* number of edges in the graph). Our traffic compiler sweeps the design space and automatically selects the best decomposition limit.

2) Clustering: In Fig. 12, we show the effect of *Clustering* on performance with increasing PE counts. *Clustering* provides an improvement over *Decomposition* since it accounts for compute and message injection costs accurately, but that improvement is small (1%–18%). Remember from Section III, that *Clustering* is a lightweight, inexpensive optimization that attempts to improve load balance and as a result, we expect limited benefits.

3) Placement: In Fig. 13, we observe that *Placement* provides as much as 2.5× performance improvement over a random placed workload as PE counts increase. At high PE counts, localized traffic reduces bisection bottlenecks and communication latencies. However, *Placement* is less effective at low PE counts since the NoC is primarily busy injecting and receiving traffic and NoC latencies are small and insignificant. Moreover, good load-balancing is crucial for harnessing the benefits of a high-quality placement.

As we can see in Figure 14, we get speedups of 0.5-13× (geomean 3.6×) for Sparse Matrix Solve benchmarks when using a high-quality placement for distributing the graph operations when compared to a random placement. Random placement performs poorly for the Token Dataflow compute model as it is unable to minimize network bandwidth. For the `s208` benchmark, random placement does provide a 2× benefit over good placement. This is because our placement algorithm currently does not specifically attempt to constrain long critical dependency chains within the PE. However for all other benchmarks, the high-quality placer is able to outperform

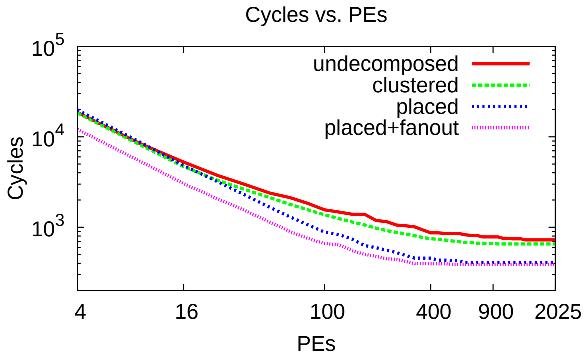


Fig. 15: *Clustering*, *Placement* and *Fanout-Routing* (`ibm01`)

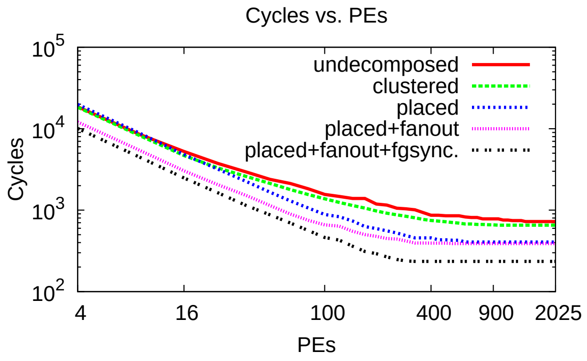


Fig. 16: *Clustering*, *Placement*, *Fanout-Routing* and *Fine-Grained Synchronization* (`ibm01`)

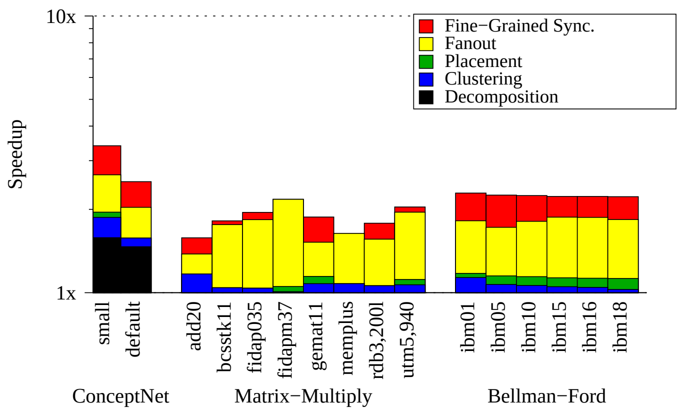


Fig. 17: Performance Ratio at 25 PEs

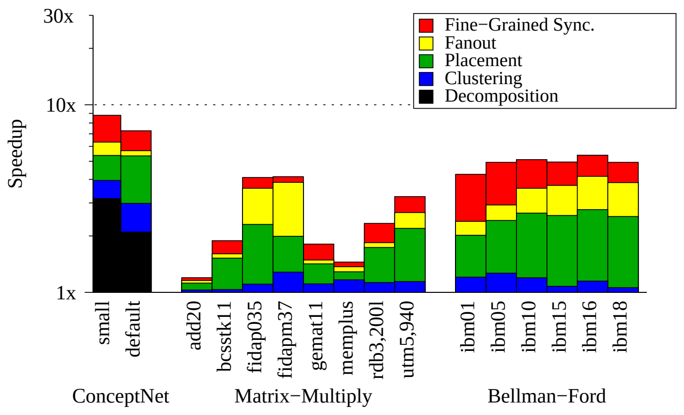


Fig. 18: Performance Ratio at 256 PEs

random placement by optimizing bandwidth. In the future, with a different placement strategy that targets latency of the computation, we expect to be able to outperform random in all cases by an even greater amount.

4) Fanout-Routing: We show performance scaling with increasing PEs for the Bellman-Ford `ibm01` workload using *Fanout Routing* in Fig. 15. The greatest performance benefit (1.5×) from *Fanout Routing* comes when redundant messages distributed over few PEs can be eliminated effectively. The absence of benefit at larger PE counts is due to negligible shareable edges as we suggested in Section III.

5) Fine-Grained Synchronization: In Fig. 16, we find that the benefit of *Fine-Grained Synchronization* is greatest (1.6×) at large PE counts when latency dominates performance. At low PE counts, although NoC latency is small, elimination of the global barrier enables greater freedom in scheduling PE operations and consequently we observe a non-negligible improvement (1.2×) in performance. Workloads with a good balance between communication time and compute time will achieve a significant improvement from fine-grained synchronization due to greater opportunity for overlapped execution.

C. Cumulative Performance Impact

We look at cumulative speedup contributions and relative scaling trends of all optimizations for all workloads at 25 PEs, 256 PEs and 2025 PEs. The relative impact and importance of these optimizations shift as a function of system size. In some cases, a particular optimization is irrelevant at a particular PE count point in the NoC design space *e.g.* fanout routing is most useful at small system sizes, placement is important at larger system sizes.

At 25 PEs, Fig. 17, we observe modest speedups in the

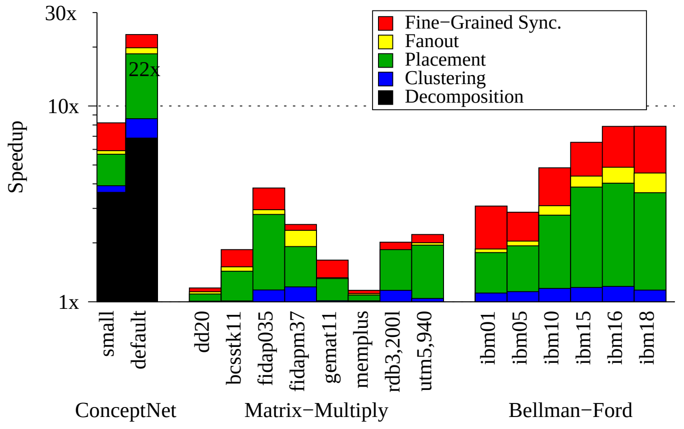


Fig. 19: Performance Ratio at 2025 PEs

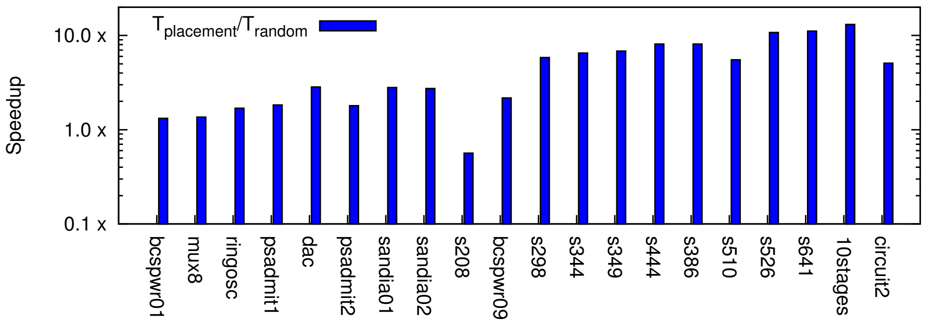


Fig. 14: Impact of *Placement* at 25 PEs
(Sparse Matrix Solve computation)

range 1.5× to 3.4× (2× mean) which are primarily due to *Fanout Routing*. *Placement* and *Clustering* are unable to contribute significantly since performance is dominated by computation. *Fine-Grained Synchronization* also provides some improvement, but as we will see, its relative contribution increases with PE count.

At 256 PEs, Fig. 18, we observe larger speedups in the range 1.2× to 8.3× (3.5× mean) due to *Placement*. At these PE sizes, the performance bottleneck begins to shift to the network, so reducing traffic on the network has a larger impact on overall performance (see previous discussion in Section V-A). We continue to see performance improvements from *Fanout Routing* and *Fine-Grained Synchronization*.

At 2025 PEs, Fig. 19, we observe an increase in speedups in the range 1.2× to 22× (3.5× mean). While there is an improvement in performance from *Fine-Grained Synchronization* compared to smaller PE cases, the modest quantum of increase suggests that the contributions from other optimizations are saturating or reducing.

Overall, we find ConceptNet workloads show impressive speedups up to 22×. These workloads have decomposable nodes that allow better load-balancing and have high-locality. They are also the only workloads which have the most need for *Decomposition*. Bellman-Ford workloads also show good overall speedups as high as 8×. These workloads are circuit graphs and naturally have high-locality and fanout. Matrix-Multiply workloads are mostly unaffected by these optimization and yield speedups not exceeding 4× at any PE count. This is because the compute phase dominates the communicate phase; compute requires high latency (9 cycles/edge from Table II) floating-point operations for each edge. It is also not possible to decompose inputs due to the non-associativity of the floating-point accumulation. As an experiment, we decomposed both inputs and outputs of the `fidapm37` workload at 2025 PEs and observed an almost 2× improvement in performance.

D. Cumulative Area and Energy Impact

For some low-cost applications (*e.g.* embedded) it is important to minimize NoC implementation area and energy. The

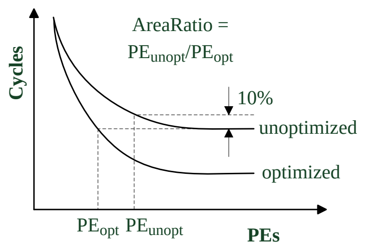


Fig. 20: How we compute area savings

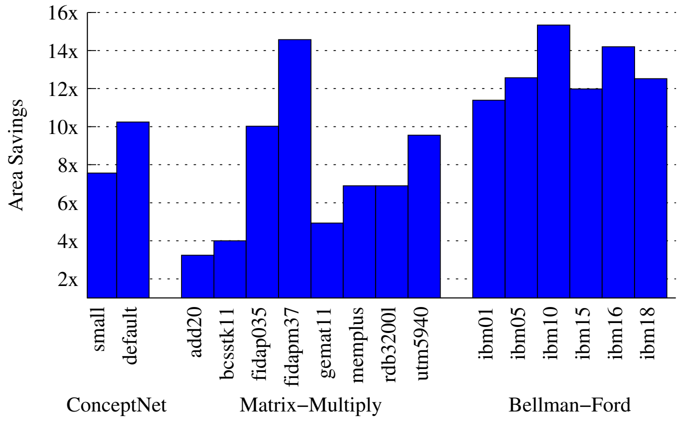


Fig. 21: Area Ratio to Baseline

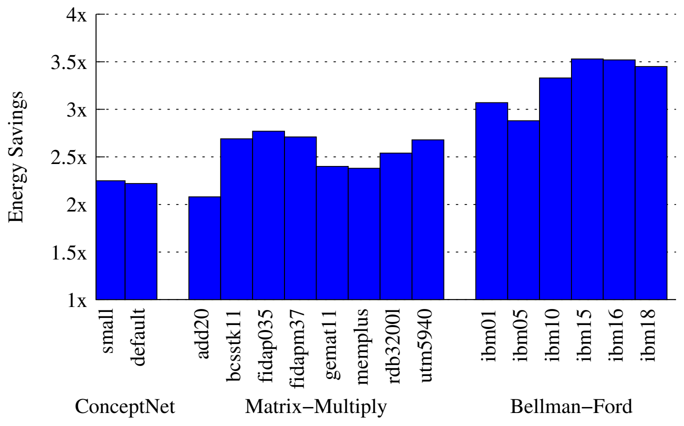


Fig. 22: Dynamic Energy Savings at 25 PEs

optimizations we discuss are equally relevant when cost is the dominant design criteria.

To compute the area savings, we first pick the smallest PE count (PE$_{unopt}$ in Figure 20) that is within 1.1× the cycle count of the best possible application performance for the unoptimized workload (the 10% slack accounts for diminishing returns at larger PE counts). For the fully optimized workload, we identify the PE count (PE$_{opt}$ in Figure 20) that yields performance equivalent to the best unoptimized case. We report these area savings in Figure 21. The ratio of these two PE counts is 3–15 (mean of 9), suggesting these optimizations allow much smaller designs.

To compute energy savings, we use the switching activity factor and network cycles to derive dynamic energy reduction in the network. Switching activity factor is extracted from the number of packets traversing the *Split* and *Merge* units of a Mesh Switch over the duration of the simulation.

$$Activity = (2/Ports) \times (Packets/Cycles)$$

In Figure 22 we see a mean 2.7× reduction in dynamic energy at 25 PEs due to reduced switching activity of the optimized workload. While we only show dynamic energy savings at 25 PEs, we observed even higher savings at larger system sizes which have even higher message traffic.

VI. Conclusions and Future Work

Large, on-chip networks that support highly-parallel, fine-grained applications will be required to handle heavy message traffic. Load-balancing, communication bandwidth, IO serialization and synchronization costs will play a key role in determining the performance and scalability of such systems. We develop a traffic compiler for sparse graph-oriented workloads to automatically optimize network traffic and minimize these costs. We demonstrate the effectiveness of our traffic compiler over a range of real-world workloads with performance improvements between 1.2× and 22× (3.5× mean), PE count reductions between 3× and 15× (9× mean) and dynamic energy savings between 2× and 3.5× (2.7× mean) for the BSP workloads. We also show speedups of 0.5-13× (geomean 3.6×) for Sparse Matrix Solve (Token Dataflow) workloads when performing a high-quality placement of the dataflow graphs. For large workloads like `cnet-default`, our compiler optimizations were able to extend scalability to 2025 PEs. We observe that the relative impact of our optimizations changes with system size (PE count) and our automated approach can easily adapt to different system sizes. We find that most workloads benefit from *Placement* and *Fine-Grained Synchronization* at large PE counts and from *Clustering* and *Fanout Routing* at small PE counts. While we have demonstrated these optimizations for a specific compute model, the techniques are applicable to an even larger space of possible compute models.

References

[1] R. K. Brayton and C. McMullen, "The decomposition and factorization of boolean expressions," in *Proc. Intl. Symp. on Circuits and Systems*, 1982.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[3] D. GreenField, A. Banerjee, J. G. Lee, and S. Moore, "Implications of Rent's rule for NoC design and its fault tolerance," in *NOCS First International Symposium on Networks-on-Chip*, 2007.

[4] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Enginering Approach*. Elsevier, 2003.

[5] D. Yeung and A. Agarwal, "Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient," *SIGPLAN Notices*, vol. 28, no. 7, pp. 187–192, 1993.

[6] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon, "GraphStep: a system architecture for Sparse-Graph algorithms," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 143–151.

[7] N. Kapre and A. DeHon, "Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs," in *International Conference on Field-Programmable Technology*, 2009, pp. 190–198.

[8] L. G. Valiant, "A bridging model for parallel computation," *CACM*, vol. 33, no. 8, pp. 103–111, August 1990.

[9] T. Davis and E. P. Natarajan, "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems," *ACM Transactions on Mathematical Software*, vol. 37, no. 3, pp. 1–17, 2010.

[10] G. Papadopoulos and D. Culler, "Monsoon: an explicit token-store architecture," *Proceedings of the Annual International Symposium on Computer Architecture*, vol. 18, no. 3a, pp. 82–91, 1990.

[11] W. Ho and T. Pinkston, "A methodology for designing efficient on-chip interconnects on well-behaved communication patterns," in *Proc. Intl. Symp. on High-Perf. Comp. Arch.*, 2006.

[12] V. Soteriou, H. Wang, , and L.-S. Peh, "A statistical traffic model for on-chip interconnection networks," in *Proc. Intl. Symp. on Modeling, Analysis, and Sim. of Comp. and Telecom. Sys.*, 2006.

[13] Y. Liu, S. Chakraborty, and W. T. Ooi, "Approximate VCCs: a new characterization of multimedia workloads for system-level MPSoC design," *DAC*, pp. 248–253, June 2005.

[14] G. Varatkar and R. Marculescu, "On-chip traffic modeling and synthesis for MPEG-2 video applications," *IEEE Trans. VLSI Syst.*, vol. 12, no. 1, pp. 108–119, January 2004.

[15] J. Balfour and W. J. Dally, "Design tradeoffs for tiled CMP on-chip networks," in *Proc. Intl. Conf. Supercomput.*, 2006.

[16] R. Mullins, A. West, and S. Moore, "Low-latency virtual-channel routers for on-chip networks," in *ISCA*, 2004.

[17] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 1025–1040, August 2005.

[18] N. Kapre, N. Mehta, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216.

[19] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, 1993.

[20] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays - FPGA '99*, pp. 37–46, 1999.

[21] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," in *Proceedings of the Asia and South Pacific Design Automation Conference*, January 2000, pp. 661–666.

[22] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," *SIGPLAN Not.*, vol. 30, no. 8, pp. 144–155, 1995.

[23] H. Liu and P. Singh, "ConceptNet – A Practical Commonsense Reasoning Tool-Kit," *BT Technical Journal*, vol. 22, no. 4, p. 211, October 2004.

[24] NIST, "Matrix market," <http://math.nist.gov/MatrixMarket/>, June 2004, maintained by: National Institute of Standards and Technology (NIST).

[25] M. deLorimier and A. DeHon, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," in *FPGA*, February 2005, pp. 75–85.

[26] N. Kapre and A. DeHon, "Parallelizing sparse Matrix-Solve for SPICE circuit simulation using FPGAs," in *Proceedings of the 8th International Conference on Field Programmable Technology*, 2009.

[27] *The Programmable Logic Data Book-CD*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, 2005.