Appearing in *Reconfigurable Communication-centric Systems on Chip* (ReCoSoC), May 17–19, 2010

An NoC Traffic Compiler for efficient FPGA implementation of Parallel Graph Applications

Nachiket Kapre
California Institute of Technology,
Pasadena, CA 91125
nachiket@caltech.edu

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

***Abstract*—Parallel graph algorithms expressed in a Bulk-Synchronous Parallel (BSP) compute model 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. 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. 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 allows 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.
- Most application 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.
- Finally, 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*.

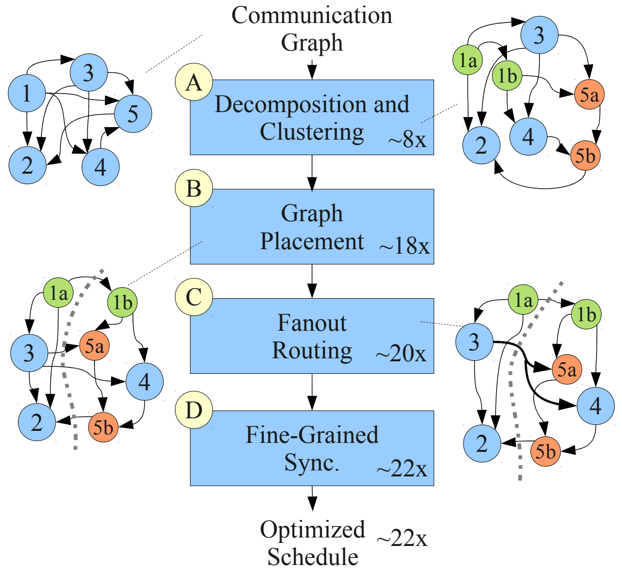

Fig. 1: NoC Traffic Compilation Flow
(annotated with `cnet-default` workload at 2025 PEs)

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 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 applications described using the BSP compute model.
- Use of communication workloads extracted from ConceptNet, Sparse Matrix-Multiply and Bellman-Ford running on range of real-world circuits and graphs.
- Quantification of cumulative benefits of each stage of the compilation flow (performance, area, energy).

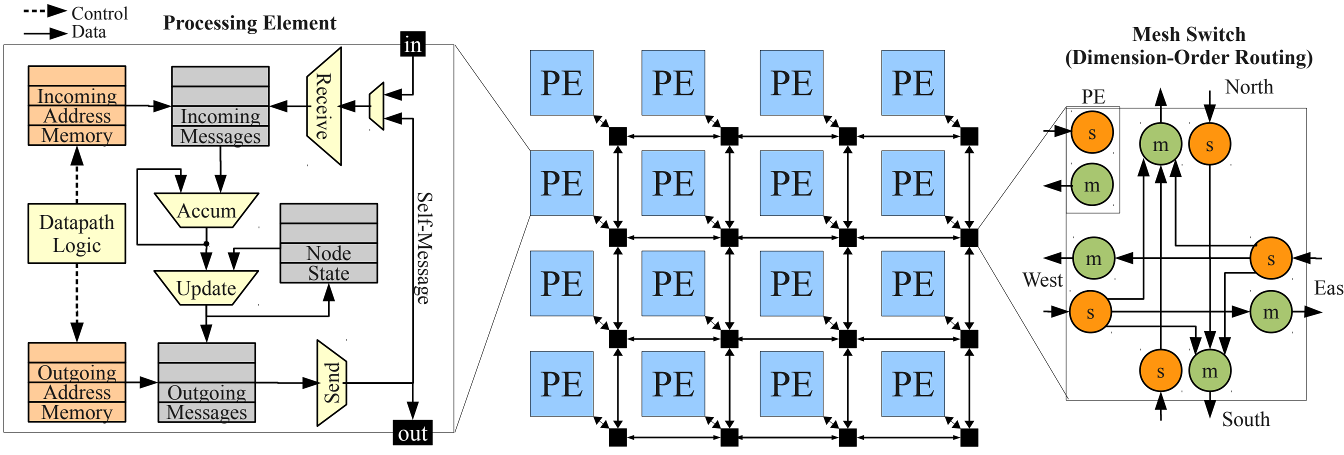

Fig. 2: Architecture of the NoC

II. Background

A. Application

Parallel graph algorithms are well-suited for concurrent processing on FPGAs. We describe graph algorithms in a Bulk Synchronous Parallel (BSP) compute model [6] and develop an FPGA system architecture [7] 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 BSP 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. Applications in the BSP compute model generate traffic with many 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.* [8]–[11]) and random workloads (*e.g.* [12]–[14]) for analysis and experiments. Statistical and random workloads may exaggerate traffic requirements and ignore application structure leading to overprovisioned NoC resources and missed opportunities for workload optimization.

In [8], the authors demonstrate a 60% area reduction along with an 18% performance improvement for well-behaved workloads. In [10], we observe a 20% reduction in buffer sizes and a 20% frequency reduction for an MPEG-2 workload. In [12], the authors deliver a 23.1% reduction in time, a 23% reduction in area as well as a 38% reduction in energy for their design. We demonstrate better performance, lower area requirements and lower energy consumption (Section V).

B. Architecture

We organize our FPGA NoC as a bidirectional 2D-mesh [15] with a packet-switched routing network as shown in Figure 2. The application graph is distributed across the Processing Elements (PEs) which are specialized to process graph nodes. Each PE stores a portion of the graph in its local on-chip memory and performs accumulate and update computations on each node as defined by the graph algorithm. The PE is internally pipelined and capable of injecting and receiving a new packet in each cycle. The switches implement a simple Dimension-Ordered Routing algorithm [16] and also support fully-pipelined operation using composable *Split* and *Merge* units. We discuss additional implementation parameters in Section IV-B. Prior to execution, the traffic compiler is responsible for allocating graph nodes to PEs. During execution, the PE iterates through all local nodes and generates outbound traffic that is routed over the packet-switched network. Inbound traffic is stored in the incoming message buffers of each PE. The PE can simultaneously handle incoming and outgoing messages. 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. We measure network performance as the number of cycles required for one epoch between barriers, including both computation and all messages routing.

III. Optimizations

In this section, we describe a set of optimizations performed by our traffic compiler.

1) 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,

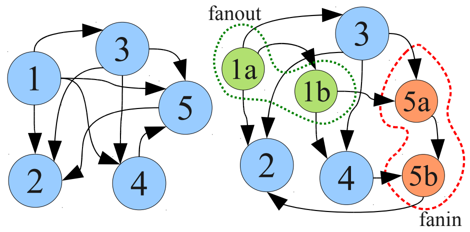

Fig. 3: *Decomposition*

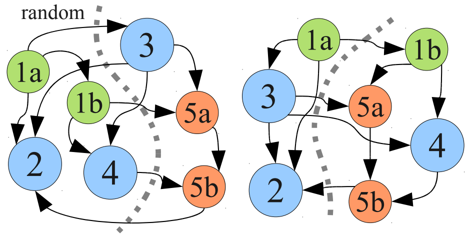

Fig. 4: *Placement*
(Random Placement vs. Good Placement)

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). *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 (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

2) 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 and receiving messages.

3) 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* 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. 4 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 better cut with 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 [17] that iteratively clusters nodes and moves the clustered nodes around partitions to search for a better quality solution.

4) 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 work, there needs to be at least two sink nodes in any destination PE. The PE will then internally distribute the message to the intended recipients. This is shown in Fig. 5. 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.

5) 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

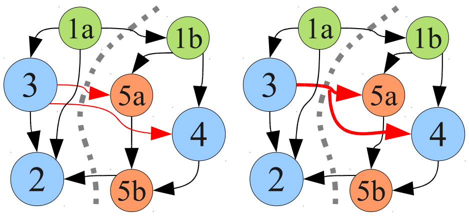

Fig. 5: *Fanout-Routing*

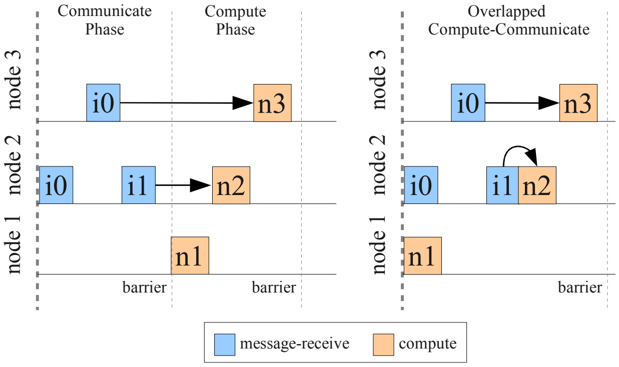

Fig. 6: *Fine-Grained Synchronization*

have been previously studied [5], [18]. 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. We show the potential benefit of *Fine-Grained Synchronization* in Fig. 6. 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.

IV. Experimental Setup

A. Workloads

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

1) ConceptNet: ConceptNet [19] 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 [7], we show a per-FPGA speedup of 20× compared to a sequential implementation.

TABLE I: Application Graphs

Graph	Nodes	Edges	Max	
			Fanin	Fanout
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
`rdb3200l`	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

2) Matrix-Multiply: 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. 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 [20]. In [21], we show a speedup of 2-3× over optimized sequential implementation using an older generation FPGA and a performance-limited ring topology.

3) Bellman-Ford: 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. 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.

B. NoC Timing and Power Model

All our experiments use a single-lane, bidirectional-mesh topology that implements a Dimension-Ordered Routing function. The Matrix-Multiply network is 84-bits wide while ConceptNet and Bellman-Ford networks are 52-bits wide (with 20-bits of header in each case). The switch is internally pipelined

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}	2
Processing Element	Latency
T_{send}	1
$T_{receive}$ (ConceptNet, Bellman-Ford)	1
$T_{receive}$ (Matrix-Multiply)	9

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 Merge	0.26 0.72	1.07 1.58	1.45 2.1	1.65 2.49	1.84 2.82
84 (Matrix-Multiply)	Split Merge	0.32 0.9	1.35 1.87	1.78 2.45	2.02 2.88	2.26 3.25

to accept a new packet on each cycle (see Figure 2). 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 to start processing a new edge every cycle. ConceptNet and Bellman-Ford compute simple sum and max operations while Matrix-Multiply performs floating-point $accumulation$ on the incoming messages. Each computation on the edge then takes 1 or 9 cycles of latency to complete (see Table II). We estimate dynamic power consumption in the switches using XPower [22]. 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 modern Xilinx Virtex-5 LX110T FPGA [22] to derive our timing and power models shown in Table II and Table III.

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. We order the optimization appropriately to analyze their additive impacts. First 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.

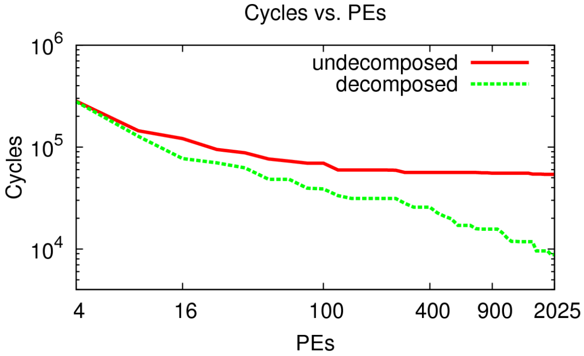

Fig. 7: *Decomposition* (`cnet-default`)

A. *Impact of Individual Optimizations*

1) Decomposition: In Fig. 7, 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 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 8 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. 9, 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. 10, we observe that *Placement* provides as much as 2.5× performance improvement over

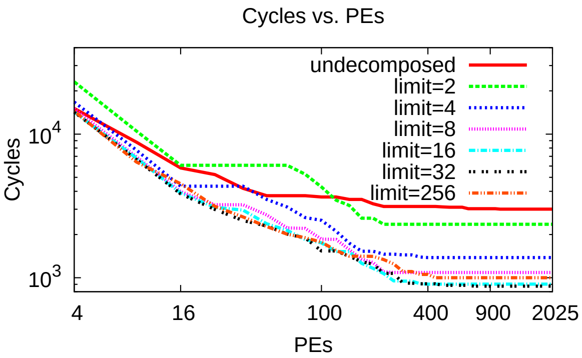

Fig. 8: *Decomposition Limits* (`cnet-small`)

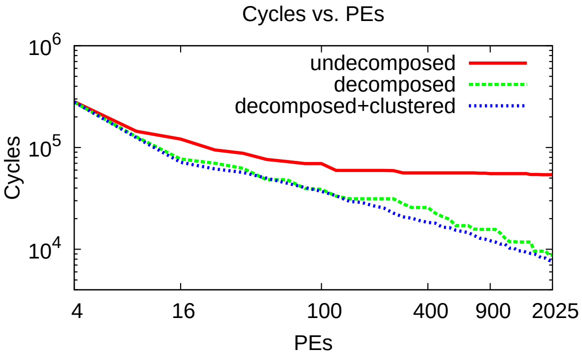

Fig. 9: *Decomposition* and *Clustering* (`cnet-default`)

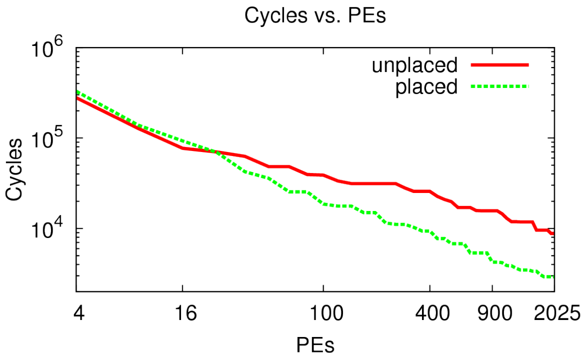

Fig. 10: *Decomposition*, *Clustering* and *Placement* (`cnet-default`)

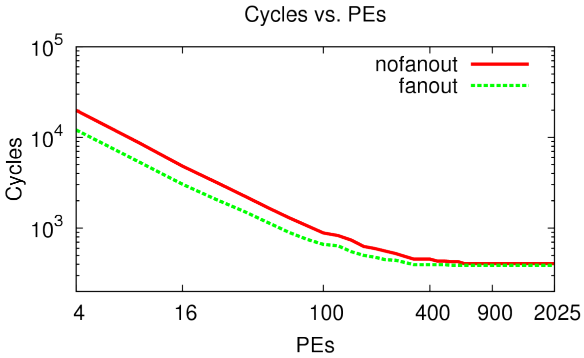

Fig. 11: *Clustering*, *Placement* and *Fanout-Routing* (`ibm01`)

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 (See Figure 15 with other benchmarks).

4) Fanout-Routing: We show performance scaling with increasing PEs for the Bellman-Ford `ibm01` workload using *Fanout Routing* in Fig. 11. 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. 12, 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.

B. 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.

At 25 PEs, Fig. 13, we observe modest speedups in the 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. 14, 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. We continue to see performance improvements from *Fanout Routing* and *Fine-Grained Synchronization*.

At 2025 PEs, Fig. 15, 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*

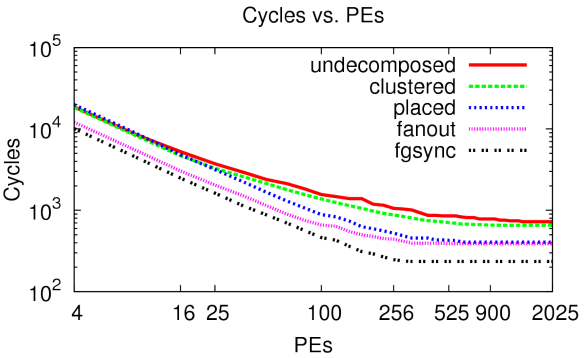

Fig. 12: *Clustering*, *Placement*, *Fanout-Routing* and *Fine-Grained Synchronization* (`ibm01`)

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.

C. Cumulative Area and Energy Impact

For some low-cost applications (*e.g.* embedded) it is important to minimize NoC implementation area and energy. The optimizations we discuss are equally relevant when cost is the dominant design criteria.

To compute the area savings, we pick the smallest unoptimized PE count that requires 1.1× the cycles of best unoptimized case (the 10% slack accounts for diminishing returns at larger PE counts (see Figure 16). For the fully optimized workload, we identify the PE count that yields performance equivalent to the best unoptimized case. We report these area savings in Figure 17. 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 18 we see a mean

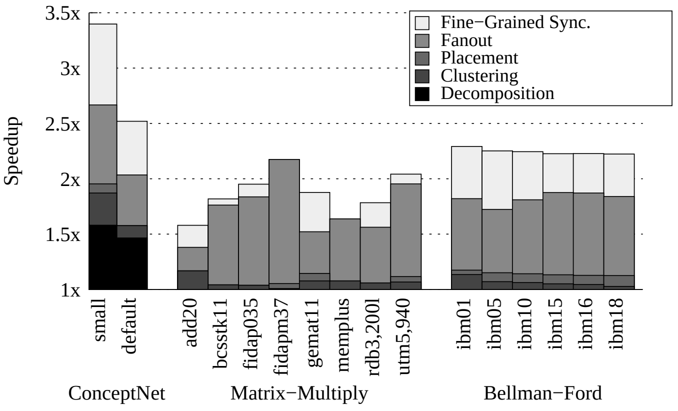

Fig. 13: Performance Ratio at 25 PEs

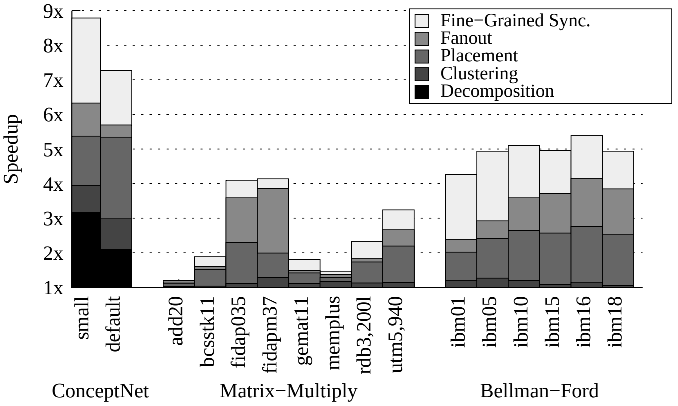

Fig. 14: Performance Ratio at 256 PEs

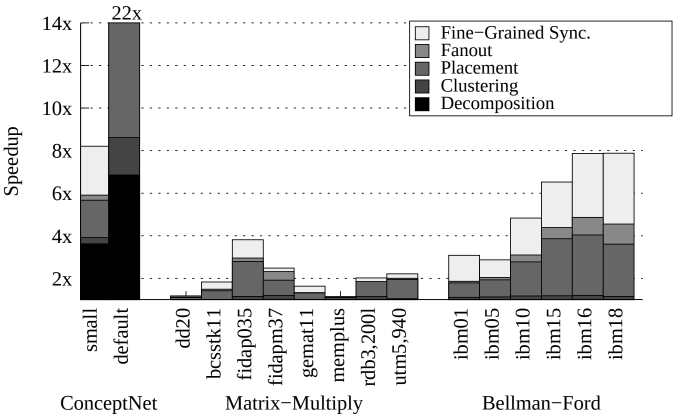

Fig. 15: Performance Ratio at 2025 PEs

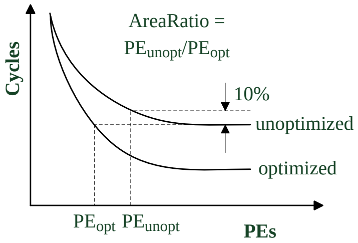

Fig. 16: How we compute area savings

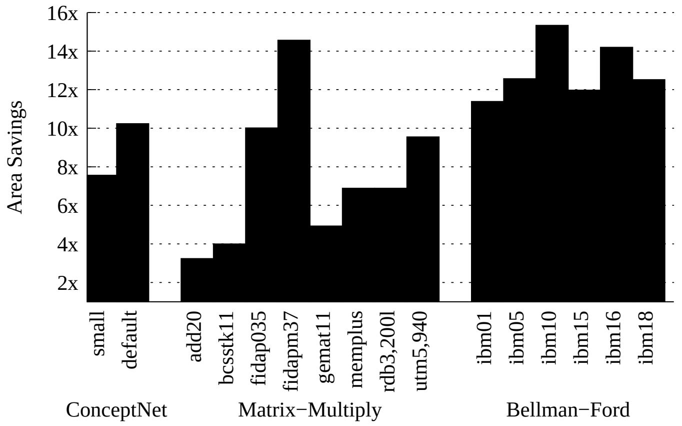

Fig. 17: Area Ratio to Baseline

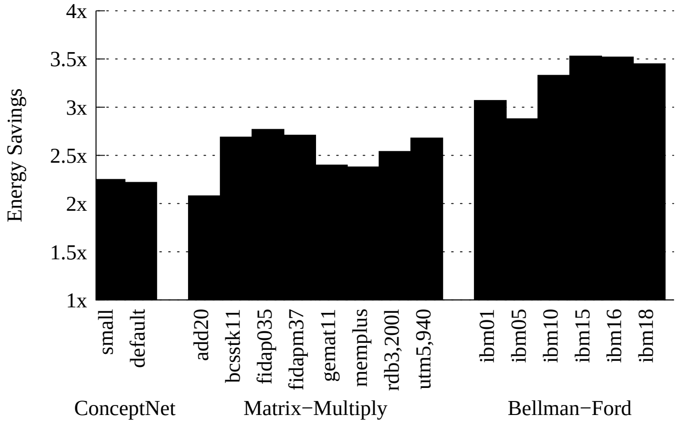

Fig. 18: Dynamic Energy Savings at 25 PEs

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.

VI. Conclusions and Future Work

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 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. The optimizations we describe in this paper have been used for the SPICE simulator compute graphs which are different from the BSP compute model. Similarly we can extend this compiler to support an even larger space of automated traffic optimization algorithms for different 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. I, III-1

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

[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. I, III-3

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

[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. I, III-5

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

[7] 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. II-A, IV-A1

[8] 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. II-A

[9] 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. II-A

[10] 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. II-A

[11] 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. II-A

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

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

[14] 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. II-A

[15] 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. II-B

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

[17] 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. III-3

[18] C.-W. Tseng, "Compiler optimizations for eliminating barrier synchronization," *SIGPLAN Not.*, vol. 30, no. 8, pp. 144–155, 1995. III-5

[19] H. Liu and P. Singh, "ConceptNet – A Practical Commonsense Reasoning Tool-Kit," *BT Technical Journal*, vol. 22, no. 4, p. 211, October 2004. IV-A1

[20] NIST, "Matrix market," <http://math.nist.gov/MatrixMarket/>, June 2004, maintained by: National Institute of Standards and Technology (NIST). IV-A2

[21] M. deLorimier and A. DeHon, "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," in *FPGA*, February 2005, pp. 75–85. IV-A2

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