Generating Synthetic Task Graphs for Simulating Stream Computing Systems ☆

Deepak Ajwani [a,1,*], Shoukat Ali[b], Kostas Katrinis[b], Cheng-Hong Li[c], Alfred J. Park[d], John P. Morrison[a], Eugen Schenfeld[c]

[a] *The Centre for Unified Computing, University College Cork, Cork, Ireland*
[b] *Exascale Systems Group, IBM Research Dublin, Dublin, Ireland*
[c] *IBM Research, TJ Watson Research Center, Yorktown Heights, N.Y. 10598*
[d] *Oak Ridge National Laboratory, Oak Ridge, TN 37831*

☆ This is an extended version of the conference publication "A Flexible Workload Generator for Simulating Stream Computing Systems" [2] and an IBM research report "Application-agnostic Generation of Synthetic Task Graphs for Stream Computing Applications" [3]

* Corresponding Author – Email: d.ajwani@cs.ucc.ie, Phone: +353-21-4205935

[1] The research work of the author is supported by an Enterprise Partnership Scheme grant from the Irish Research Council on Science, Engineering and Technology and IBM

Abstract

Stream-computing is an emerging computational model for performing complex operations on and across multi-source, high-volume data flows. The pool of mature publicly available applications employing this model is fairly small, and therefore the availability of workloads for various types of applications is scarce. Thus, there is a need for synthetic generation of large-scale workloads to drive simulations and estimate the performance of stream-computing applications at scale. We identify the key properties shared by most task graphs of stream-computing applications and use them to extend known random graph generation concepts with stream computing specific features, providing researchers with realistic input stream graphs. Our graph generation techniques serve the purpose of covering a disparity of potential applications and user input – Our first "domain-specific" framework exhibits high user-controlled configurability while the second "application-agnostic" framework only takes the number of vertices as an input.

Keywords: Stream computing systems, Workload characterization, Computational task graphs, Synthetic stream-computing graphs

1. Introduction

Stream processing is an important emerging computational model for performing complex operations on and across multi-source, high volume data flows. Today there are many different applied areas of stream computing such as audio/video applications [18, 23], continuous database queries [1], intelligent transportation systems [6], analysis of social networks [13], real-time analysis of financial and medical data [14].

Large-scale simulations will play a major role in the process of assessing the performance and for proof-of-concept prototyping of processing, networking and operating system hardware and software for stream-processing systems. For example, a researcher could investigate algorithms for partitioning graphs that represent stream computing systems. This could, in turn, help the researchers to see design alternatives that they might not have otherwise considered during the development of new partitioning algorithms.

Ideally, such simulations will be based on the real application data (e.g., [8]). However, the pool of publicly available mature real-world stream-computing applications is fairly small, and therefore, the availability of workloads for such simulations is scarce. As such, there is a need for generating large-scale synthetic input data (e.g., computation task graphs, traffic matrices). Realistic synthetic workloads will also be a significant aid in planning the capacity that needs to be placed on various resources of a stream computing system, such as memory sizes, processing power, network interface card buffer sizes and switch port numbers. To guarantee reliability and reproducibility of results, a standard framework for generating such workloads needs to be evolved.

We present a methodology for producing synthetic workloads for stream computing systems. A particular focus of our work is on generating computational task graph (where the vertices represent the various operations being performed on the streams and the edges represent the data-streams), along with data-rates and behaviors of streams.

A major challenge in designing generators for stream-computing graphs is

to determine the key properties that a typical instance of a stream-computing graph should satisfy and to gather the relevant statistics to fill up the specifications. To this end, we first record a set of characteristic properties that are believed to be shared by the task graphs of large stream-computing applications. We then propose two generation frameworks that emulate the recorded properties. Our first framework is highly flexible as it allows the users to synthesize various workloads by specifying a range of values for many different parameters that model the characteristics of stream computing task graphs from their application domain. Our second framework is "application-agnostic" in the sense that it relieves the user from choosing a domain-specific combination of parameters. It takes only one input parameter – the number of nodes – and emulates the key properties of stream-computing task graphs, in general.

An important consideration in our design of generation techniques had been to ensure that our techniques can be used to generate large streaming task-graphs (with more than a million edges) in a reasonable time (less than an hour with a single thread on a typical workstation). Thus, we avoid computationally expensive primitives such as maximum weight perfect matchings. However, this makes it difficult to generate a random graph that meets the user specifications and/or the characteristic properties exactly and therefore, we rely on heuristics that approximate the requirements.

For both frameworks, we show that the generated workloads match the characteristic properties of stream-computing applications fairly well. In addition, the "application-agnostic" framework also matches the domain-specific properties prescribed by the user.

The flexibility offered by our first framework allows the user to specify the degree distribution, the mix of different types of vertices and various other characteristics of the task graph. While there is a lot of work (e.g., [7, 9, 27]) on generating random undirected graphs with prescribed degree distribution and satisfying other constraints, little is known for directed graphs. We address this challenge by designing a new procedure that directs the undirected graphs so as to satisfy most of the directed constraints.

The rest of the paper is organized as follows. Section 2 describes the key properties of the task graphs from stream processing systems. Section 3 describes the various configurable features of our application-specific task graphs representing the stream-computing workloads. Section 4 presents the implementation details for generating the specified task graphs, including our novel direction assigning procedure. Section 5 validates how well the generated task graph matches the input specifications. Section 6 describes our application-agnostic stream-computing task graph generator and Section 7 shows the degree to which the generated graphs satisfy the key properties identified in Section 2. Section 8 presents the related work. Finally, Section 9 concludes with a summary and future directions.

2. Key Properties of Stream-Computing Task Graphs

In this section, we record the key properties of the stream-computing graphs. Based on various discussions with researchers who have first-hand experience with streaming applications, we conjecture that most stream-computing graphs will satisfy these properties. However, at the moment, the pool of publicly available streaming applications is too small to obtain statistically relevant measurements supporting these properties.

1. stream-computing graphs are very sparse. Since each edge represents a high-volume, continuous data-flow, a large number of edges imply communication of massive amounts of data. This is quite likely a result of a poor design choice.

   Due to their sparse character, stream-computing graphs have low average degree. Nonetheless, they can have some joins/splits with large fan-ins/fan-outs.

2. There are no vertices with more than 1 in-degree and more than 1 out-degree. All vertices fall into one of the following three types.

   - Filters: Vertices with in-degree 1 and out-degree 1. In general, filters can do any kind of data transformation including but not restricted

to sampling, filtering, sliding window computations. A special case, identity filters, merely pass the data as they receive it.

- Split: Vertices with in-degree 1 and out-degree greater than 1. The splits are subdivided into following categories: Copy splits that copy the input stream to output streams; Round-robin or If-else distributors that distribute the input stream into output streams.
- Join: Vertices with in-degree greater than 1 and out-degree 1

3. **Statistics on Kernel Mix**: The following are some observations on the relative proportion of kernel mix in stream-computing applications and their input and output stream rates.
   - A large majority of the vertices are filters.
   - For the StreamIT framework, it was observed that around 35% of splits are copy splits – they copy the input stream to output streams [24, 25]. The remaining splits are mostly distributing splits – they distribute the input stream (in a round-robin way, based on value of elements etc.) into the output streams.
   - Most joins merge the input streams in some way, i.e., their output stream rate is the sum of input stream rates.

   Even though the StreamIT framework restricts the stream-computing graphs to be split-parallel graphs, we believe that many properties observed for the streaming primitives in this framework also hold for entire applications. In particular, this should hold for the statistics on kernel mixes.
4. Stream-computing graphs are mostly acyclic. Note that there may be cycles (feedback loops) involving control signals that are short data-flows, but we do not model them as edges in our computational task graph. The edges in our definition of stream-computing graphs strictly correspond to continuous, high-volume data streams.
5. For any vertex pair (x, y), all paths from x to y have roughly the same

length, where the length is defined as the number of edges in the path independent of the weights on the constituent edges.

6. In the stream computing applications we studied, computationally intensive tasks are usually performed towards the end of the computation process after the initial kernels have sampled and reduced the data volume significantly. In other words, the weights on nodes closer to the sink are significantly higher than those that are closer to the source. On the other hand, the weight of edges that reflect the data-flow volume decreases as we traverse from sources to sink in the directed acyclic graphs. In particular, the filter vertices in the early part of the computation process significantly reduce the data rate.
7. There are more splits than joins close to the sources and there are more joins than splits closer to the sinks.

Our graph generation frameworks attempt to emulate these properties. Our user-configurable framework focuses on meeting the key requirements prescribed by the user from her particular application-domain. The expectation is that the user specification will match the properties identified above and we show in Section 5 that under such a setting, the framework matches the specifications very well. In contrast, our application-agnostic framework aims to generate large random graphs that directly satisfy the above properties.

3. User-Configurable Workload Features

In this section, we identify various features characterizing stream computing workloads (and particularly its task graph), and specify them in the context of the workload generator.

We perceive that a typical workload executing on a large stream computing system would have thousands of applications executing on it. A subset of these applications will communicate with each other. Within one application, there might be thousands of communicating kernels. This expectation means that a typical synthetic workload will look like a forest of weakly connected

components. Each weakly connected component (WCC) represents one user application. It is also expected that a given application might have several instances running in the system. This may happen, for instance, because a given application might be invoked repeatedly (in parallel) to operate on different subsets of the same data or with different parameters (e.g., for sampling, optimization).

In our workload generation framework, a user can specify the number of WCCs and configure the key features of each WCC. In addition, the user can also specify certain global features for the entire stream-computing graph (forest of WCCs): (a) the total number of vertices that should be present in the final stream graph; (b) the average number of vertices that should be present in a WCC; (c) the maximum number of sources that could drive a WCC; and the maximum number of sinks that could be driven by a WCC.

A vertex in a WCC represents a stream-computing kernel while an edge represents the flow of data-stream from one kernel to another. Next, we describe the various features of a WCC that can be configured by the user. Note that our generator may not be able to generate a WCC that *exactly* matches the user specification, but it tries to ensure that it comes close.

3.1. Numbers of Sources and Sinks

The workload generator allows the user to specify the number of sources and sinks for a given WCC. A user can also assign a probability distribution and its parameters to sample the values of the number of sources and sinks.

3.2. Communication Patterns

The generator gives the user two ways of specifying communications among the kernels of a given WCC:

1. **Relative Proportions of Kernels Categories** A user can specify percentages for the three types of vertices (i.e., join, split, and filter), as well as the percentages of joins and splits with different input and output edges, respectively. However, these specifications should be realizable and

in particular, they should satisfy some boundary conditions, such as the number of out-edges that will result from the user specification should be equal to the number of in-edges and vice-versa (as every out-edge has to be an in-edge of some vertex).

We expect the user to specify the kernel mix for all vertices in the WCC and we try to match it as close as we can. If the kernel mix is under-specified, then all remaining vertices are assumed to be filters.

2. **Degree Distribution** The user could specify a degree distribution for the vertices. This may be in the form of a degree sequence (degrees of vertices in sorted order), average degree, maximum degree or a probability distribution of degrees (e.g., power-law degree distribution).

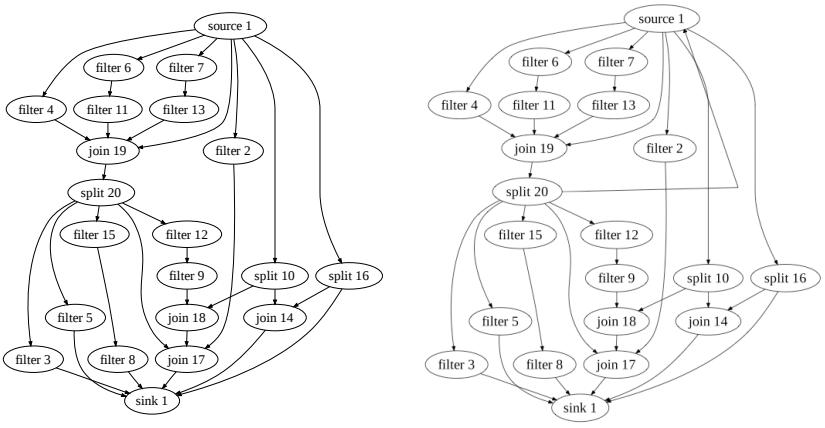
3.3. Feedback Arcs in the WCCs

We perceive that a typical stream computing workload could have several instances of feedback in it. To simulate such feedback, our workload generator allows users a method to introduce back-edges (also referred to as feedback arcs). Given a directed acyclic graph G, an edge $u \rightarrow v$ is a back-edge if there is a path from v to u in G. We refer to the longest path from v to u as the back-length of the feedback arc $u \rightarrow v$. A user specifies the back-length of all feedback arcs in the WCC.

A single back-edge can result in many different cycles. For instance, the WCC in Figure 1(b) was produced from the WCC in Figure 1(a) by introducing the back edge `(split 20,source 1)`. There are many paths of varying length between `source 1` and `split 20`, which results in many cycles, specifically {`source 1, filter 7, filter 13, join 19, split 20, source 1`}, {`source 1, join 19, split 20, source 1`}, {`source 1, filter 4, join 19, split 20, source 1`}, and {`source 1, filter 6, filter 11, join 19, split 20, source 1`}. Note that none of the cycles so created is larger than back-length plus one.

We believe that in many real streaming applications, cycles are caused by feedback arcs that can potentially lead to multiple cycles. We try to emulate

this by focusing on introducing the appropriate feedback arc and not so much on creating only the required cycle. Also, the requirement of a constraint on maximum length of cycles frequently arises in partitioning problems where the quality of a partition is considered better if all cycles are contained within the same computing node or a set of tightly integrated computing nodes. To evaluate if a particular algorithm is indeed doing that, a reasonable testing procedure would test for a graph of fixed size, but increasing size of the largest cycle. Thus, our generator can be potentially useful for such testing heuristics as well.



(a) (b)

Figure 1: The WCC in (b) was produced from the acyclic WCC in (a) by adding a cycle of length 5.

3.4. Workload Composition

The join vertices within an application may have different firing rules: An output might be produced (a) only when all inputs have had new data sets (b) as soon as any input has had a new data set, or (c) as soon as a particular input has a new data set. Our workload generator hands the choice to the user to specify what percentage of the multi-input kernels will have a given firing rule.

3.5. Sensor Stream Modeling

We perceive that each application might be driven by several sensor streams. The specificities of the sensor rates and stream sizes will heavily depend on the characteristics of the application that is modeled by a WCC of the workload graph. Serving the objective of high configurability, the generator provides for a wide range of sensor stream modeling options corresponding to many different application types. It is then the responsibility of the end user to pick the configuration that closely matches the actual sensor stream patterns for the application of interest. In the following, we elaborate on various features of sensor streams modeled by the workload generator.

3.5.1. Input Stream Rates

Users can specify the rate pattern of "primal" streams entering a given WCC. The primal streams are the ones entering a WCC from the outside world, and in the computational graph abstraction, they are denoted by vertices labeled as "sensors." As part of the workload synthesis, the generator allows the user the ability to specify, for each sensor, the distribution and the related parameter values (e.g., average, variance etc.) governing the rate of data stream it produces.

3.5.2. Push vs. Pull Sensor Streams

By the term push sensor streams we refer to streams that are "pushed" incrementally to memory and as such it is not guaranteed that a stream will be available for processing by the source kernel, every time the latter becomes idle. Such a model corresponds to applications involving capturing of real-time data/events, such as surveillance or network monitoring. Instead, pull sensor streams guarantee the existence of a stream to process, every time the host processor attempts to have a stream input into the kernel graph of a specific task. The pull streams can be used to model tasks that process stored data, such as data mining over bulk data. Note that the generator does not limit the user to using one of the two models for a specific kernel graph, allowing thus

the combination of push and pull sensor streams, e.g., for modeling a network monitoring application that compares real-time traffic against stored malicious traffic patterns.

To avoid any potential confusion, the terms "push" and "pull" for sensor streams refer to the way sensor streams are loaded on the streams memory, not the method used to feed them to source kernels. It is the source kernel's program code that fetches streams from the streams data, whenever such are available.

3.5.3. Intra-sensor dependence of push streams

When examining push sensor streams flowing along a specific sensor, we differentiate between the following two cases with reference to statistical independence between any pair of streams.

- **Push streams with statistically independent data inter-arrival times**, such as for example inventory tracking data or aggregation of customer-input to a call center over long-time intervals (in general, streams that aggregate data input from a large number of independent sources).
- **Push streams with statistically dependent data inter-arrival times**: streams exhibiting burstiness fall in this category, for instance road-traffic management data or any type of logged data exhibiting temporal dependence (e.g., Web site visits on a daily basis). Constant-rate push streams are a special case within this category.

3.5.4. Inter-sensor dependence of push streams

For push sensor streams, the generator allows the user to model the following.

- **Sensor streams that are pairwise statistically independent**
- **Sets of sensor streams that are statistically dependent**, as for example the stream pairs of a pair of synchronized audio and video sources or the streams corresponding to the network traffic of a group of network segments that share a common bottleneck link.

3.5.5. Fixed versus variable stream element size

While many applications will work with fixed sized stream elements (e.g., video frames of constant resolution), others may generate variable sized stream elements (e.g., monitored network packets). To provide the capability of stream-size dependent performance evaluation, the generator hands the user the ability to accordingly model the stream size along a sensor stream, by specifying either a fixed stream size value or a distribution (and its parameter values) that will be sampled to randomly generate stream size values.

3.6. Workload Heterogeneity

It is likely that the systems adopting the stream computing model will have a variety of hardware computational nodes. For each such node, our approach allows the user to specify how greatly and with what properties (e.g., probability distribution) the execution times of the different kernels vary for any given node. The user can specify task heterogeneity by specifying the coefficient of variation that must be observed by the simulated execution times. The user can specify one of the following two distributions for the estimated kernel processing times: Pareto and Gamma. The Gamma distribution was used because of its generality and the Pareto distribution was used because many researchers have shown that task processing times are often Pareto distributed or otherwise have high variability (e.g., [15, 22]).

3.7. Syntax for our Generator

Our generator tool takes input both as a command line option and from an external file. Number of sources, number of sinks, total number of vertices, seeds for random generators, required back-lengths, the user input and output file name can all be specified as command line arguments. A typical input file looks like Table 1. There are three entries: the second entry corresponds to the indegree, the third to the outdegree and the first entry denotes the number of vertices with this in-degree and out-degree. The output of our code is a dot file of a graph closely matching the user specification. Also, we

present statistics regarding the generated graph. The issues of inter-dependence of different streams, stream sizes and workload heterogeneity are handled by a post-processing tool that takes a separate input.

# Vertices	Indegree	Outdegree
1	0	1
1	1	0
128	1	2
128	2	1

Table 1: User specifications for a particular kernel mix. Here, the user wants a graph with one source, one sink, 128 splits with out-degree 2 and 128 joins with in-degree 2.

4. Framework for Generating User-Configured Stream-Computing Graphs

In this section, we outline our framework for generating the workloads based on the user specifications. In particular, we focus our attention to generating a directed graph with a prescribed degree distribution or kernel mix, a given number of sources and sinks and having feedback edges of prescribed back-lengths.

To generate (weakly) connected directed graphs with a given degree distribution, we propose various techniques. These techniques meet the user specification to varying accuracy and they offer different levels of design flexibility using different degree of randomness. In some techniques, we first generate random undirected graphs satisfying the corresponding undirected constraints and then assign the directions to meet the specified (out- and in-) degree distribution. In some other techniques, we directly generate the (acyclic) directed graphs with the required degree distribution. While the former set of techniques offer more design flexibility and provide better trade-offs between randomness and meeting user specifications, the later set of techniques are simpler and faster.

We invoke various instances of these different generating methods multiple times using parallel threads. In case a user specified a probability distribution for

the number of sources, sinks, total number of vertices, degree distribution and back-lengths of feedback edges for the required task graph, we pass randomly sampled values to the threads. In the end, we select an arbitrary graph that was succesfully generated based on the specifications for the corresponding thread.

4.1. Generation Techniques Based on Undirected Graphs

In this section, we present our techniques for generating directed graphs with a specified degree distribution based on first creating an undirected graph, then assigning directions to edges and eventually fixing the number of sources and sinks. A big advantage of these techniques is that this framework allows us to focus on generating the undirected graphs properly, which is often more important than directed graph for algorithms like graph partitioning. Furthermore, there is a large body of literature on generating random undirected graphs with various properties. We can leverage this to offer more design flexibility in our framework by simply substituting the undirected graph generation procedure and integrating those properties. For instance, substituting our undirected graph generation by the generators in [21] allows us to tune the clustering coefficient while preserving the degree distribution.

4.1.1. Generating the Undirected Graph

The problem of generating undirected graphs with prescribed degree distribution has been studied both from a finite degree sequence [7, 27] and an asymptotic point-of-view [9]. We employ the graph generator of [27] to produce the basic undirected graph. The degree sequence fed to the graph generator of [27] is calculated based on the communication requirements specified in Section 3.2.

The algorithm of [27] first creates $u.degree$ stubs for each vertex u. Then the algorithm selects a vertex v that has at least one stub left. Among all vertices not adjacent to v with at least one stub left, let w be the vertex with most stubs. The algorithm inserts an edge from v to w and removes a stub each from v and w. Once a simple graph (with no self-loops, no parallel edges) is generated

this way, the algorithm connects the different components by performing edge swaps. The resultant graph is then randomized using further edge swaps that leave the graph simple and connected.

4.1.2. Assigning Directions

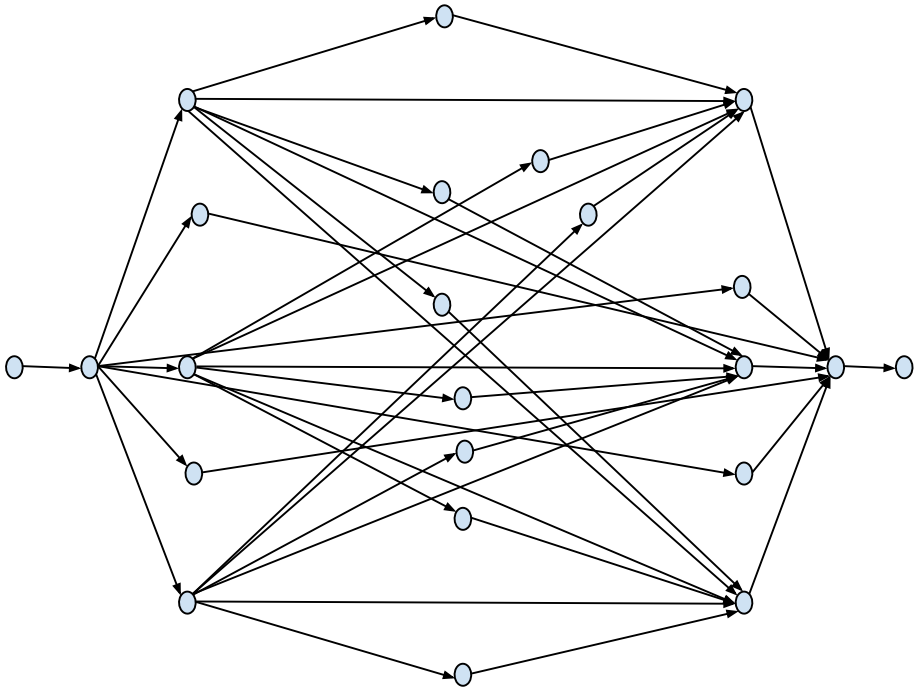


Figure 2: A directed graph realizing the user specification of a source, a sink, thirteen filters, three 6-way splits, three 6-way joins and a 7-way split and a 7-way join.

In general, it is not guaranteed that a correct direction assignment exists that can satisfy the user specification on the number of kernels if the undirected graph matches the appropriate undirected degree constraints. We consider the following example to illustrate this point. Let the user specification be one source (with out-degree 1), one sink (with in-degree 1), thirteen filters, one split with out-degree 7, three splits with out-degree 6, three joins with in-degree 6 and one join with in-degree 7. The configuration is clearly realizable as shown in Figure 2. The corresponding undirected degree constraint is to generate two

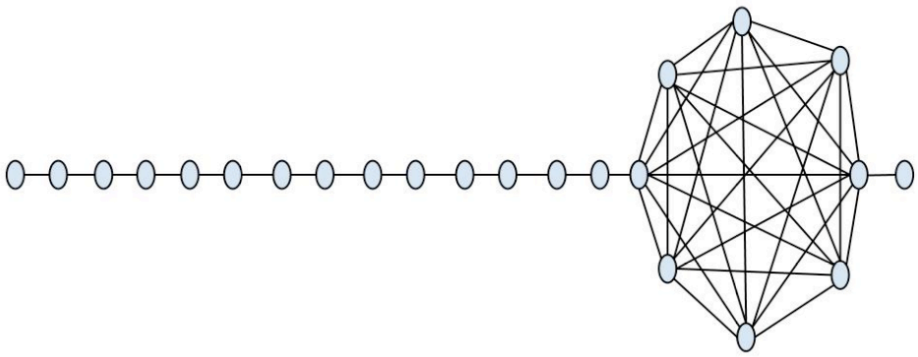


Figure 3: An undirected graph that can not be assigned directions to satisfy the user specification of a source, a sink, thirteen filters, three 6-way splits, three 6-way joins and a 7-way split and a 7-way join.

nodes with degree 1 (a source and a sink), thirteen nodes with degree 2 (filters), six nodes with degree 7 (three splits and three joins) and two nodes with degree 8 (a split and a join). An undirected graph matching these constraint is shown in Figure 3. There is no way of assigning directions to this graph to satisfy the constraints on the kernels as we prove next.

Theorem There is no assignment of direction to edges in Figure 3 that matches the user specification

Proof The undirected graph has a clique of eight vertices. By the specification, four of them need to be splits. The subgraph of the clique induced by these four splits will have six edges. However, each split can have at most one in-edge and assigning directions to these six edges between the split vertices will ensure that the sum of the in-degrees of the four splits is at least six, leading to a contradiction. Therefore, no direction assignment is possible for the given specification □

The above argument proves that it is not possible to design a direction assignment procedure that will always match the kernel specification *exactly*. Even in case when such an assignment is possible, it can result in a large number of directed cycles (that may violate the specification on the cycle lengths). As

such, we focus on heuristics that can match the user requirement *approximately* and are fast in practice.

Our first and simpler approach to make the above graph directed is to assign directions independently and uniformly at random. Afterwards, we visit each vertex, and declare it as one of the aforementioned three types depending upon the numbers of inputs and outputs. In case there is a vertex x with multiple inputs and multiple outputs (thereby not falling into any one of three types of vertices: filter, join and split), we transform it into two vertices u and v such that all inputs of x become inputs of u, all outputs of x become outputs of v, and there is an edge from u to v. This transformation may increase the number of vertices in the graph.

This process, however, does not take into account the relative proportions of different types of vertices, as in Section 3.2(1). And even for the case where the user specified the communication requirements only in the form of undirected degree distribution, as in Section 3.2(2), it is not clear how well the above transformation preserves the user-specified degree distribution. With high probability, each vertex x with large degree will be multiple-input multiple output vertex under random edge assignment and therefore, x will be divided into a split-join pair u, v with degree of x getting distributed between u and v.

We now provide an alternative procedure that addresses these shortcomings. The algorithm starts with an undirected graph and assigns directions to all edges. At any intermediate stage, the graph has both directed and undirected edges with degree refering to the total number of incident edges (including both directed and undirected edges) and indegree and outdegree refering to total number of directed in-edges and out-edges, respectively.

We assume that the sources and sinks have degree 1. This can be easily ensured by inserting virtual sources and sinks with degree 1 and connecting them to the actual sources and sinks. Once the graph is generated, the virtual sources and sinks can be removed. The algorithm first divides the degree 1 vertices into sources and sinks based on their prescribed numbers and assigns their incident edges accordingly. It then attempts to assign the directions based

on the following rules:

- A vertex with degree 2 and indegree (outdegree) 1 should have the incident undirected edge outgoing (incoming).
- A vertex with indegree (outdegree) greater than 1 and outdegree (indegree) equal to 1 should have all incident undirected edges incoming (outgoing).
- A vertex with indegree (outdegree) equal to degree minus 1 should have the remaining incident undirected edge outgoing (incoming).

If it can't apply these rules to any vertex of the graph, then it makes a probabilistic decision to determine if a vertex with degree greater than 2 should be a join or split. This decision is based on the number of degree d joins ${}^{d-1}J^1$ and splits ${}^1S^{d-1}$ that still need to be created. Note that ${}^{d-1}J^1$ (${}^1S^{d-1}$) is simply the difference between the user-specified number for degree d joins (splits) and the number of degree d joins (splits) in the graph based on the current assignment of edges.

Consider a set S of vertices that have at least one directed and one undirected edge. It can be shown that if the corresponding input graph is connected, then the set S will be empty only if all edges have been assigned directions.

The probabilitic decision first selects the vertex $v \in S$ that has the maximum number of directed edges. If v has indegree greater than 1, it is declared a join. Otherwise, if v has outdegree greater than 1, it is declared to be a split. If none of these cases hold, then we decide v to be a split with a probability ${}^1S^{d-1}/({}^1S^{d-1} +^{d-1} J^1)$ and join otherwise.

Consider the case when v is declared to be split. If v has an indegree at least 1, all incident undirected edges are directed away from v. Otherwise, we randomly select an incident undirected edge to be an in-edge and direct the remaining edges away from v. We treat the case of deciding v to be a join analogously.

It can easily be proved that our algorithm always terminates. Once the

process terminates, all edges will be assigned a direction. Note that there may still be a few vertices with multiple inputs and multiple outputs. However, the number of such vertices for this procedure is likely to be significantly less than that for random edge assignment. These vertices are transformed into a split-join pair, similar to the random edge assignment procedure.

4.1.3. Sources and Sinks

Let N_i^{srcs} be the number of sensors sampled from a user supplied distribution for a given WCC i. To ensure that the number of sensors in the WCC i is exactly N_i^{srcs}, we proceed as follows. We identify the vertices with zero in-degree, and declare these vertices as tentative sources. If the number of tentative sources is less than N_i^{srcs}, we discard this WCC from inclusion in the final graph. Otherwise, we arbitrarily pick N_i^{srcs} tentative sources and label them as sources. For each remaining tentative source vertex v, we add an edge originating from a randomly chosen source to v. The case for sinks is dealt in an analogous manner.

4.2. Generating Directed Graphs Directly

In this section, we present our techniques for generating the directed graphs directly. These techniques first create a number of out-stubs and in-stubs for each vertex, equal to the out-degree and in-degree of the vertex and then connect all the out-stubs to in-stubs. The edges in the generated graph correspond to the connections between the appropriate out-stubs and in-stubs.

Our first approach is to connect the in-stubs and out-stubs randomly (i.e., map each out-stub to a random in-stub). This is quite simple and faster than our undirected graph based approaches. However, it generates graphs with larger number of cycles of varying lengths (including 2-cycles $u \to v \to u$). The removal of these cycles causes more deviation from the user specification than our approach of assigning directions to the undirected graph (refer to Section 5.2 for more details).

Our next approach meets the user specification perfectly, but is deterministic. This means that for a given degree distribution, it always generates the same graph instance, rather than selecting an instance at random from all the graphs satisfying the prescribed degree distribution.

The deterministic approach considers the nodes in the following order: Sources, Splits (in decreasing order of their degree), Filters, Joins (in increasing order of their degree), Sinks. For instance, if a user specified a source, a sink, 4-way split, 2-way split and two 3-way joins, they will be considered in the following order: source, 4-way split, 2-way split, 3-way join, 3-way join and then a sink. We arrange both the out-stubs and in-stubs in the above order and map the i^{th} out-stub to the i^{th} in-stub generating the digraph G. There is an edge (u, v) in G if and only if an out-stub of node u is mapped to an in-stub of node v.

Theorem The digraph G satisfies the user specified degree-distribution and is acyclic with no self-loops.

Proof Sketch The out-degree and the in-degree of a node u in G equals the number of out-stubs and in-stubs respectively of u. The number of stubs are generated based on the prescribed degree distribution. Thus, the degree distribution is met perfectly.

To see that the resultant digraph is acyclic and has no self-loop, we prove that the order of nodes as follows defines a topological ordering: Source, Splits (in decreasing order of degree), Filters, Joins (in increasing order of degree), Sinks. In other words, we will prove that all edges in the digraph G go from an earlier node to a later node in this order, establishing that there are no self-loops and no cycles.

Let $OT(i) = j (IT(i) = j)$ (for out-stub and in-stub to topological order mapping) iff the i^{th} out-stub (in-stub) corresponds to a node u that is at j^{th} position in the above defined order. Next, we show that $OT(i) < IT(i)$, thereby proving that all edges go from a lower ordered node to a higher ordered node in the defined order.

Let $TO(j) = i (TI(j) = i)$ (for topological order to out-stub and in-stub mapping) iff a node u that lies at the j^{th} position in the topological order

has its first out-stub (in-stub) at position i. By definition, $OT(TO(j)) = j$, $TO(OT(i)) \leq i$, $IT(TI(j)) = j$ and $TI(IT(i)) \leq i$. Also note that OT and TO are monotonically increasing.

Owing to the way we have ordered these nodes, $TO(j) \geq TI(j+1)$ for all indices j. Putting everything together, we get $TO(OT(i)) \leq i \implies TI(OT(i)+1) \leq i \implies OT(i)+1 \leq IT(i) \implies OT(i) < IT(i)$ □

The above approach can be randomized by selecting an in-stub with a probability inversely proportional to the number of unmatched in-stubs before it. However, increasing the degree of randomness in this procedure (by increasing the probability of later in-stubs to be selected) results in increasing number of cycles and self-loops, whose removal results in larger deviation from the user specification.

4.3. Inserting Feedback Arcs of Prescribed Back-Lengths

Our approach starts by removing all pre-existing cycles. Removing a minimum number of edges to make a digraph acyclic (minimum feedback arc set problem) is known to be NP-hard [16]. We therefore propose a simple heuristic that works well in our setting.

We introduce a new dummy source vertex and insert edges from the dummy source to all source vertices of the original graph. We then employ a depth-first search starting from the dummy source, to identify and delete all back-edges, thereby removing all cycles.

Next, we insert feedback arcs of prescribed back-lengths $L_1, \ldots, L_k$. First, we order the vertices in a topological order such that all edges go from lower ordered vertex to a higher ordered vertex. We then use dynamic programming to compute for each vertex y, the length of longest path from any real source s to y. To insert a feedback arc of length L_i, we search for a vertex pair (x, y) such that x has an outdegree 1, y has an indegree 1 and there is a path of length $L_i - 1$ from x to y. We then insert the back edge (y, x) that results in a cycle of length L_i while ensuring that no vertex has multiple input and multiple output. If there is no such vertex pair (x, y), we abort the procedure and restart from

scratch.

This procedure ensures that upon successful completion of the graph generation process, all feedback arcs of prescribed lengths will be added. However, as noted in Section 3.3, this process may introduce a larger number of cycles.

5. Validation

By virtue of its design, our graph generation framework has at least the number of user-specified vertices and the feedback edges of required back-lengths. Therefore, in this section, we focus on how well the generated graph meets its kernel mix, how many extra vertices are added because of transforming a multi-input, multi-output vertex into a split-join pair and how big a graph we can generate in a reasonable time. We employ the following methodology for this purpose: We specify various kernel mixes and for each mix, we generate more than 100 graphs based on that specification. Then for each vertex type (such as split with degree d), we compare the user specification to the average number generated by our approach. Most stream-processing graphs in practice are sparse and they have a large number of filter vertices. We use this observation to specify the type and degree of all vertices (described in Table 2) for our experiments. In order to ensure that the graph is sparse, we specify most joins and splits to have small degree.

We treat Specification 1 as the standard specification; Specification 2 is designed to have a larger number of filter vertices; Specification 3 is designed to have assymetric split join specification and Specification 4 has a few vertices with very high degree. All our specifications require acyclic graphs, which is the default option when no back-length is prescribed.

As we will show later, our algorithm matches the user specification closely for graphs with large number of filter vertices (such as in Specification 4). We therefore decided to focus on harder cases such as Specification 1 and 3 with fewer filter vertices to stress-test our algorithm.

Table 2: Kernel mixes used for Validation

Type	Indeg	Outdeg	S. 1	S. 2	S. 3	S. 4
Source	0	1	4	4	4	4
Sink	1	0	4	4	4	4
Filter	1	1	3200	6800	3200	7400
Split	1	2	800	200	1300	100
Split	1	3	800	200	1100	100
Split	1	4	800	200	150	100
Split	1	500	0	0	0	4
Join	2	1	800	200	900	100
Join	3	1	800	200	1300	100
Join	4	1	800	200	150	100
Join	500	1	0	0	0	4

Table 3: Average Results for Generated Graphs

Type	Indeg	Outdeg	S. 1	S. 2	S. 3	S. 4
Source	0	1	4.0	4.0	4.0	4.0
Sink	1	0	4.0	4.0	4.0	4.0
Filter	1	1	3563.1	6780.9	3700.4	7204.0
Split	1	2	855.1	225.7	1248.2	176.4
Split	1	3	813.2	203.5	901.5	109.9
Split	1	4	558.8	180.3	112.2	84.6
Split	1	500	0	0	0	2.0
Join	2	1	824.4	242.9	924.3	203.7
Join	3	1	597.2	170.5	878.5	98.9
Join	4	1	490.5	142.2	91.0	97.1
Join	500	1	0	0	0	0

5.1. Kernel Mix for Undirected Graph Based Techniques

As shown in Table 3, the generated graphs meet the user specifications to varying degree for the kernel mixes described in Table 2. For specifications with

larger number of filter vertices (as is common in streaming applications), the requirements are met satisfactorily. In particular, the overall structure of the task graphs was always very close to user expectations. For instance, if a user specified most vertices to be filters, the user did indeed get a long narrow graph and if the user specified some large degree splits and joins, she infact got a short and wide graph.

Our approach meets the user specifications for total number of vertices, sources and sinks very closely. For Specification 1, the minimum, average and maximum number of vertices were 8008, 8012.5 and 8040, as against the user specification of 8008 vertices. The generated graphs almost always had the correct number of sources and sinks.

Before the cycle removal part, the specifications for the graphs generated by our approach are within 5% of the user specification. However, in general, our directed graphs have a large number of cycles and our heuristic for removing these cycles creates significant deviation from the user specification. For Specification 1, our approach first generated a directed graph with 3159.3 filters, 826.2 2-way splits, 801.6 3-way splits, 797.3 4-way splits, 824.8 2-way joins, 802.4 3-way joins and 797.2 4-way joins. But once the cycles are removed, the 4-way split reduces to 558.8 and the 4-way joins reduce to 490.5.

Among the kernel types, the maximum error was reported for 4 to 1 joins (38.6% less than user specification for Spec. 1 and 28.9% less for Spec. 2). The fact that high degree joins get more affected by cycle removal than other vertices is an artifact of our heuristic for removing all cycles.

We managed to get a particularly poor performance from our algorithm by specifying an extreme dense case, particularly designed to elicit such behavior. It consisted of 400 joins with degree 400 and 400 splits with degree 400 and a single source and sink. In order to match this specification, the generated undirected graph had to be a complete bipartite graph with 400 vertices on each side. A random undirected graph matching the degree distribution is highly unlikely to be bipartite. This resulted in very large errors and the generated graph was very far from the user specification. However, such a specification is highly unlikely

to be reflective of any stream computing application as it pertains to very dense graphs and such a high amount of data flow in a streaming application is an indicator of some design flaw.

We also investigated the success rate of the heuristic for inserting a cycle of a given length. We defined the success rate as the number of trials in which the heuristic successfully found the required cycle to the total number of trials conducted. We experimented on a smaller scale version of Specification S.3, where the numbers of all kernel types (except sources and sinks) were 1/10 of the numbers for S.3 in Table 2. For cycles lengths 5, 10, 16, 20 and 25, we found the success rates 59%, 64.5%, 72%, 87% and 88%, respectively.

5.2. Kernel Mix for Generating Directed Graphs Directly

Table 4 shows the result of generating the directed graphs directly by randomly mapping the in-stubs and the out-stubs, on the specifications defined in Table 2. For Specification 1, results for 3-way and 4-way joins (495.0 and 409.8) are significantly worse than that for our technique (597.2 and 490.5) based on undirected graphs. The number of sinks is significantly larger (617.1 as opposed to 4 in Specification 1) than for our previous approach which matches the user specification. The increase in the number of sinks is because of the removal of a larger number of feedback edges. Also, this approach completely failed to satisfy the user requirement of vertices with degree 500 in any of our 10,000 trials. Rather, it produces vertices whose degree ranges from 350 to 430. Our approach managed to have 2.0 splits with degree exactly 500. Overall, these results (averaged over 1000 runs) clearly deviate more from the user specification than our approach based on directing the undirected graph.

5.3. Running Time

The actual running time of the generation process depends on the number of times the procedure aborts from various stages and restarts. However, as long as the graph is not too minutely specified and the user specifications reflect stream-computing applications, our generation process is successful with few

Table 4: Results for generating directed graphs directly

Type	Indeg	Outdeg	S. 1	S. 2	S. 3	S. 4
Source	0	≥ 1	5.4	5.9	5.3	5.1
Sink	≥ 1	0	617.1	274.1	575.7	10.8
Filter	1	1	3575.1	6691.4	3746.1	7394.8
Split	1	2	883.1	207.2	1320.0	101.9
Split	1	3	804.7	205.2	848.8	102.3
Split	1	4	515.9	172.3	99.9	94.6
Split	1	500	0	0	0	0
Join	2	1	701.9	195.7	738.3	101.8
Join	3	1	495.0	141.7	703.2	102.0
Join	4	1	409.8	114.5	70.5	94.5
Join	500	1	0	0	0	0

aborts. The individual steps of our generation framework are quite fast, as they do not use the computationally expensive primitives of perfect machings or maximum weight matchings. It took about 4 seconds to generate the graph given in Specification 1 in Table 2, and about 90 seconds to generate a graph that was 10 times larger on a quad-core machine with 2.67GHz processor, 8MB Cache, 12GB RAM and running a Linux operating system. To generate a graph where the specifications were 100 times those of Specification 1 in Table 2, our approach required around 50 minutes. This graph had around 1.3 million edges!

5.4. Shape of the Graph

In our framework, there are various ways to control the shape of the generated graph. For instance, specifying a large back-length ensures that the resultant graph has long paths and specifying splits and joins with high fan-out/fan-in can result in directed acyclic graphs of large width. Nonetheless, there is a need for better mechanisms to control the shape of the graph, particularly because the graph properties like width and graph diameter can vary widely for the same kernel mix. For example, for the specification in Table 1, one

graph may combine them as a chain where a pair of splits (s_1, s_2) is followed by a pair of joins (j_1, j_2) (with edges $s_1 \to j_1$, $s_1 \to s_2$, $s_2 \to j_1$, $s_2 \to j_2, j_1 \to j_2$) to form an acyclic graph with maximum width 3. Another graph satisfying the specification can have all splits before all joins, resulting in a large width of 64.

6. Application Agnostic Streaming Graph Generator

In this section, we describe the general framework for generating the stream-computing graphs emulating the above properties. As our generation framework is application-agnostic, the only user parameter it takes is n – the number of vertices in the generated graph. The output is a graph in either the dot format [12] or in the format of 10th DIMACS implementation challenge [11], which is also the format for partitioning libraries such as METIS [17]. It can generate both the directed and the corresponding undirected versions of the graph.

Our framework first generates the "core" of the streaming graph with $\Theta(n^{\frac{1}{3}})$ vertices. To create this core, we first generate an acyclic series-parallel multi-graph with $\Theta(n^{\frac{2}{3}})$ edges and then add $\Theta(n^{\frac{1}{3}})$ random edges, preserving the acyclicity of the multi-graph. This ensures that the final graph has a big series-parallel subset, though it is not series-parallel itself.

In order to generate a series-parallel multigraph, we start with two vertices and $O(n^{\frac{1}{3}})$ edges from one to another. We then repeatedly add vertex pairs till we have $\Theta(n^{\frac{1}{3}})$ vertices. Adding a vertex pair (x, y) is done by selecting an existing edge (u, v) uniformly at random, removing (u, v), adding $(u, x), (y, v)$ a single time and the edge (x, y) with a multiplicity l. The number l is chosen between 1 and $2n^{\frac{1}{3}}$, uniformly at random.

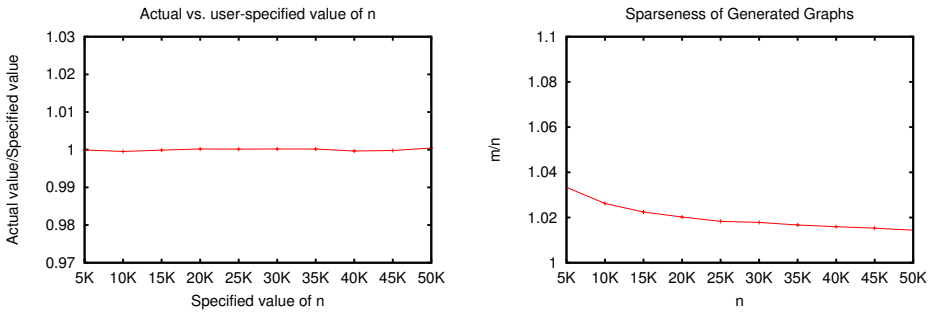
Next, we decompose the vertices with multiple in and multiple out-edges into a split-join pair. For each vertex v with indegree and outdegree greater than 1, we decompose it into a split vertex x and a join vertex y such that all in-edges to v become in-edges to x, all out-edges from v become out-edges from y and there is an edge from x to y. In the resultant graph, let $L(v)$ be the longest path

from a source vertex (with in-degree 0) to $v \in V'$ and let $D(e) = L(v) - L(u)$ for an edge $e = (u, v)$. We then replace each edge $e = (u, v)$ by a path $P(e)$ from u to v of length proportional to $D(e)$ consisting of newly created filter vertices. This ensures that in the resultant graph, all paths between two vertices have roughly the same length. As we show in Section 7, this also ensures that a large majority of vertices in the graph are filter vertices. We then sort the edges in the resultant graph and remove duplicates.

We then assign weights to vertices and edges. Let $L'(v)$ be the longest path to v in the generated topology. We assign the weight of each vertex v to be $L'^2(v)+1$. Next, we divide the splits and joins into sub-categories and assign weights to edges. Based on the statistics observed in [25] for StreamIT benchmark sub-routines, we mark 35% of splits as copying splits and the remaining as distributor splits. For the copying splits, the weight of output edges is the same as the weight of the input edge. For the distributor splits, the input weight is equally divided among the output edges. All joins add the weights of in-edges to the out-edge. The weight of the out-edge of a filter v is half that of its in-edge with probability $1/(\alpha \cdot L'(v) + 1)$ (for a small constant $\alpha < 1$) and equal to its in-edge with the remaining probability. For the source vertices with a single out-edge, we assign it a weight of 1. For a source vertex v with multiple out-edges, we declare the weight of out-edges to be 1 with probability 0.35 and $1/out\_degree(v)$ with probability 0.65. The constant α is chosen such that the edge weights decrease with the length of longest path to the tail of the edge. As shown in Section 7, such values of α do exist.

7. Validation of Application Agnostic Streaming Graph Generator

In this section, we report our experiments for generating graphs with varying number of specified vertices, averaged over 100 runs, and compare their properties with those listed in Section 2. Figure 4(a) shows that the ratio of actual number of vertices and the specified number of vertices is always very close to 1. For a given value of 10,000 vertices, we obtained an average of 9995.4 vertices,



(a) Actual vs. specified value of n (b) Sparseness of the generated graphs

Figure 4: Statistics on the number of vertices and edges in the generated graphs

thereby getting very close to the user specification. Figure 4(b) shows that in the generated graphs, the ratio of number of edges to number of vertices is close to 1. For 10,000 vertices, the average number of edges is 10257.5, which implies that the generated graphs are extremely sparse (with an average out-degree of 1.026) satisfying property 1. Since we decompose the multi-input and multi-output vertices into split-join pairs and introduce only filter vertices afterwards, we do not have any vertex with more than 1 in-degree and 1 out-degree in our graph, satisfying property 2.

The kernel mix in the generated graphs also satisfies the statistics observed in property 3. As Figure 5 demonstrates, a large majority of vertices (more than 99%) are filters while splits and joins only constitute a tiny fraction of the total number of vertices. In graphs generated with a specification of 10,000 vertices, there are 9933.03 filters, 29.79 joins and 30.59 splits out of the total number of 9995.41 actual vertices. There is only 1 source vertex in all the graphs. Out of the 30.59 splits, 11.99 (39.2%) are copy splits and 18.6 are distribute splits. All joins are merging joins.

The generation process ensures that the generated graph is acyclic (property 4).

In order to verify that the property 5 is satisfied by our generated graphs, we measure the difference between longest and shortest path from a source to an internal vertex. Consider a vertex pair (u, v) with a big path difference.

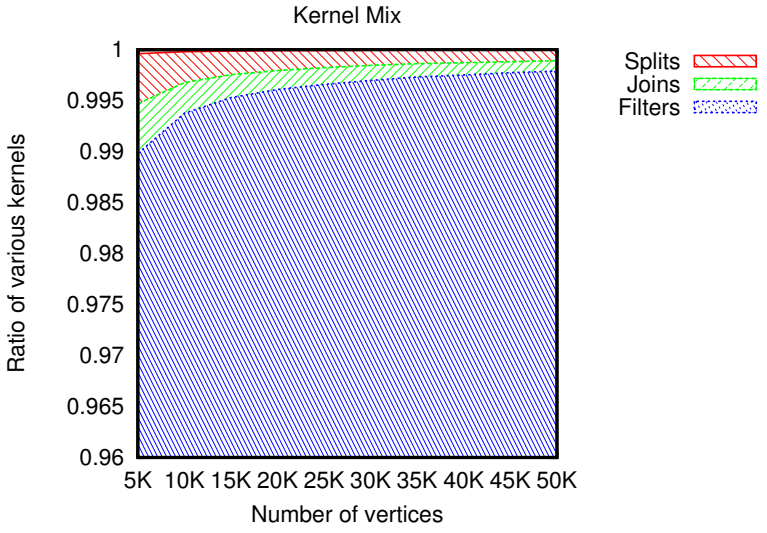
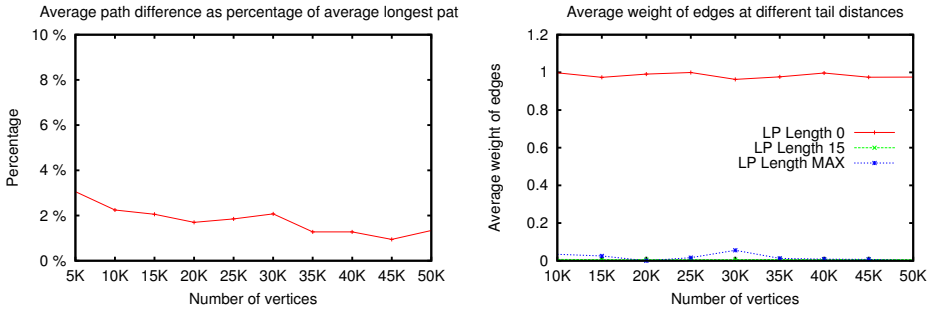


Figure 5: Kernel mix in the generated graphs

The longest path from a source to v has to be lengthier than the path from source to u followed by the longest path from u to v, while the shortest path has to be shorter than the path from source to u and shortest path from u to v. Thus, the difference between longest and shortest path to v will be greater than the maximum difference between the lengths of various paths from u to v. We observe that over all vertices in graphs generated for 10,000 vertices, the average difference between longest and shortest path is only 1.64, compared to the average longest path length of 73.11. This implies that the average path difference between vertex pairs in these graphs is smaller than 1.64. Figure 6(a) shows that over the entire range of experiments, the average difference between longest and shortest path is always a very small fraction of the longest path length.

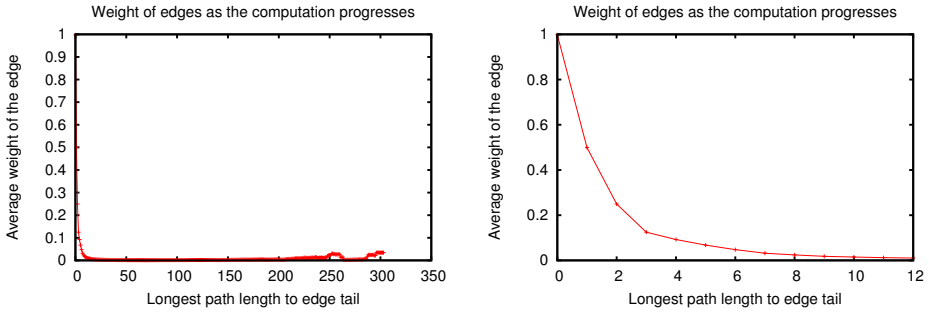
To verify property 6, we first define the tail distance of an edge (u, v) to be the length of the longest path from a source in the DAG to the tail u. Intuitively, the tail distance measures where the edge belongs in the computation process. A smaller tail distance means that the edge is very close to the source and a



(a) Average difference between the longest and the shortest path

(b) Average weight of edges for different tail distances for varying number of vertices

Figure 6: More statistics on the generated graphs

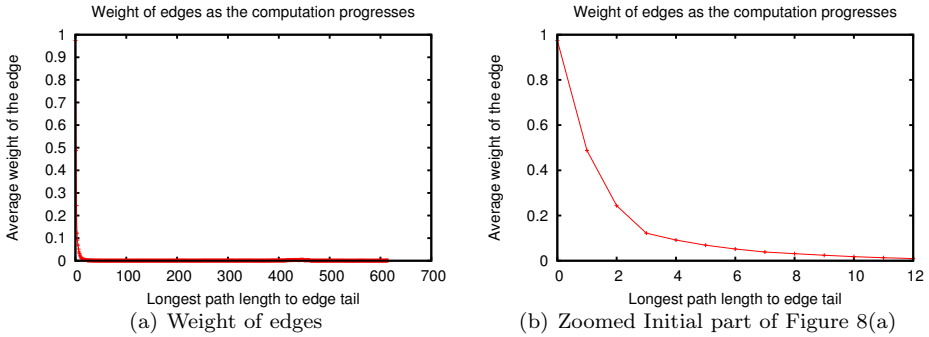


(a) Weight of edges

(b) Zoomed Initial part of Figure 7(a)

Figure 7: Weight of edges as computation progresses for 10000 vertices

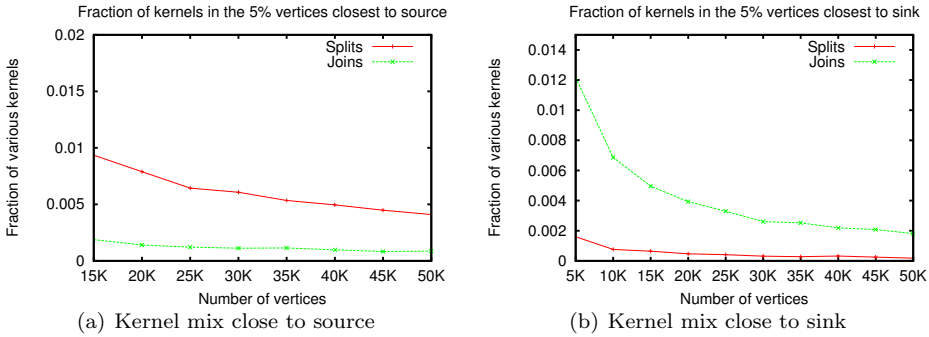
larger tail distance means that the data-flow corresponding to the edge happens at a later stage in the computation process. Figures 7(a) and 8(a) show how the weight of edges varies with the tail distance for 10,000 and 50,000 specified vertices, respectively. The weight of edge represents the total weight of all the edges with the same tail-distance in all the 100 runs of this experiment, divided by the number of such edges. As depicted in Figures 7(b) and 8(b) (which show the early part of Figure 7(a) and 8(a) in greater detail), the averaged weight drops significantly, very early in the computation process and then stays at around 1% of the early weight. The larger number of joins towards the end aggregate the weights of their in-edges and this leads to a slight increase in the edge weight, but the weight still stays below 5% in most cases. Figure 6(b) shows



(a) Weight of edges (b) Zoomed Initial part of Figure 8(a)

Figure 8: Weight of edges as computation progresses for 50000 vertices

that the trend remains the same over the entire range of experiments. Thus, our generated graphs also satisfy property 6. Note that the shape of the curve can be partly controlled by the parameter α that determines the probability of a filter vertex to reduce the data-rate. The curves in Figures 7(a) and 8(a) are obtained using $\alpha = 0.25$.



(a) Kernel mix close to source (b) Kernel mix close to sink

Figure 9: Kernel mix close to source and sink

As for property 7, we sorted the vertices by their shortest path from the source and considered the first 5% vertices. In graphs generated for 10,000 vertices, we found that 1.2% of close-to-source vertices were splits and only 0.2% were joins as compared to 0.3% splits and 0.29% joins overall. Figure 9(a) shows that for different values of n ranging from 5000 to 50,000, the number of splits close to source vertex is significantly higher than the number of joins. Similarly,

we sorted the vertices by their shortest path from the sinks and considered the first 5% vertices. In graphs generated for 10,000 vertices, we found 0.6% of close-to-sink vertices to be joins and 0.08% to be splits, thereby showing that in the generated graphs, there are significantly more splits than joins closer to sources and more joins than splits closer to sinks. Figure 9(b) shows the same trend for different values of n.

8. Related Work

There are several research efforts that deal with generating large-scale graphs such as World Wide Web graphs, the internet graphs, and cellular network graphs. However they are not fully related to this work because those efforts do not address creation of features related to stream computing, specifically communication templates like split and join, the relative proportion of different types of software kernels, creation of a given number of sources and a given number of sinks of data streams, possibility of cycles of a certain length, possibly dependent behavior for data streams that come from sensors and different firing rules for different multi-input application. Below we discuss some of the more relevant research.

The most relevant work is that of [10] where the authors describe GGen, an implementation of several classical task graph generation methods used in the scheduling domain. The generation methods they analyze are the Erdos-Renyi method, $G(n, m)$ method, layer-by-layer method, fan-in/fan-out method, and random orders method. For each method, they emphasize important graph properties that may influence scheduling algorithms. None of the graph generation methods discussed in [10] will work for stream computing systems without being significantly modified.

In the context of collecting the properties of the stream-computing graphs, a very related work is the characterization of streaming primitives in the StreamIT benchmark suite [25]. Note that in the StreamIT benchmark suite, the stream-computing graphs are restricted to be series-parallel. It is not clear if the prop-

erties of these basic primitives also holds for the entire application and whether the series-parallel restriction affects other observed properties as well. Also, this limitation to series-parallel graphs "makes code often unnatural and sometimes infeasible" [24]. We therefore do not restrict our generation process to series-parallel graphs. Nonetheless, we do use their statistics on the kernel mix.

The research in [5, 8] presents a method called SWORD to generate data streams of different kinds, including voice, image, video, text, etc. The focus of their work is on modeling the content of the data streams. Our focus is however on generating a task graph along with rates and behaviors for data streams feeding that task graph.

In [20], the authors describe a workload generator, termed StreamGen, which may be used to create a set of distributed services interacting via an application-level overlay network. Both StreamGen and our generator output labeled kernel graphs and communications thereof. However, StreamGen focusses on streaming applications characterized by human interaction, such as data streams occurring in FTP download services. Our focus is on large stream-computing systems and applications involving real-time analysis and computation such as that of financial and medical data, where the input comes from sensor streams and there is little or no human interaction.

The work in [26] studies the patterns of user interactions with streaming media servers. Once again this is not directly related to our work. It could however be the case that some particular input to a stream computing system might have the same behavior as the accesses made to a stream media server.

In [19], the authors claim that even though non-periodic bursts are prevalent in workloads of large scale applications, the existing workload models do not predict such non-periodic bursts well. They argue that including such bursts is especially important in a petabyte-scale distributed data management system. Our workload generator allows incorporating the models from [19] in the generation of sensor behavior, as given in Section 3.5.

In [28], the authors present procedures for obtaining reusable operator or kernel level resource usage information from profiling data to predict resource

usage when several operators might be fused for placement on a given host. They talk about several techniques to overcome measurement errors from the profiling data collection. Though the "workload characterization" as described by them is very different from our goal of synthetic workload generation, we note that profiling of real stream-computing systems can assist a user to configure the various parameters of our workload generator. It can also help identify new features to be incorporated in our generation framework.

9. Conclusion and Directions for Future Research

The paper presents two frameworks for generating the computational task graphs of stream computing systems, with very different configurability options – Our first framework has several design features that are programmable by the user. These include (a) number of sources and sinks, (b) relative proportion of kernel categories or degree distribution, (c) cycles of given lengths, (d) firing rules that will be applied to multi-input kernels, (e) types of sensor streams, and (f) heterogeneity of the computational requirements of kernels. Our second framework is "application-agnostic" and only takes the number of vertices as an input.

Ideally, we would like to combine these frameworks offering a user the ability to specify "domain-specific" values of configuration parameters and in case no parameters are configured, return a graph generated by the "application-agnostic" framework as default. This requires a deeper integration of the two approaches.

Our graph generators are being used for various simulations involving stream-computing applications. In particular, the "application-agnostic" framework has been accepted as a benchmark in DIMACS implementation challenge on graph partitioning [11] and has been used for evaluating the performance of a system that includes partitioning, routing and mapping of stream-computing applications on an architecture with reconfigurable optical circuit switch [4].

Both of our frameworks emulate a set of properties that have been identified

for large real stream-computing applications. Since the number of publicly available large (in terms of the graph size and not in terms of the data-volume being processed) stream-computing applications is small, we are unable to perform an independent validation of these properties. As the streaming applications grow in popularity, it might be possible to validate these properties in the future.

In our user-configurable graph generation approach, there are various ways to control the shape of the generated graph. Specifying a large-cycle length forces the graph to have long paths and specifying splits and joins with high fan-out/fan-in can increase the cardinality of the largest anti-chain. Ideally, we would like to have better mechanisms to control such properties. A joint degree distribution (e.g., specifying how many edges exist between splits of different kinds or from filters to joins of various types) can capture these and many more graph properties much better than a simple degree distribution based on kernel mix. Generating directed graphs with prescribed joint degree distribution remains an important and challenging direction for future research.

Acknowledgements

The authors would like to thank Dilma M Da silva, Qi Liu, Yoonseo Choi, Abhirup Chakraborty and Rolf Riesen for many helpful discussions and valuable feedback on an earlier version of the draft. The authors would also like to thank the anonymous reviewers of MASCOTS for their helpful comments in improving the conference submission and by extension, parts of this manuscript.

References

[1] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, The VLDB Journal 12 (2003) 120–139.

[2] D. Ajwani, S. Ali, K. Katrinis, C.H. Li, A.J. Park, J.P. Morrison, E. Schenfeld, A flexible workload generator for simulating stream computing sys-

tems, in: Proceedings of the 19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2011, pp. 409–417.

[3] D. Ajwani, S. Ali, J.P. Morrison, Application-agnostic generation of synthetic task graphs for stream com puting applications, Technical Report RC25181 (D1107-003), IBM Research, 2011.

[4] D. Ajwani, S. Ali, J.P. Morrison, Graph partitioning for reconfigurable topology, in: Proc. of the 26th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'12), IEEE Computer Society, 2012.

[5] K. Anderson, J. Bigus, E. Bouillet, P. Dube, N. Halim, null Zhen Liu, D. Pendarakis, SWORD: Scalable and flexible workload generator for distributed data processing systems, in: Winter Simulation Conference, pp. 2109–2116.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A.S. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear road: a stream data management benchmark, in: VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB Endowment, 2004, pp. 480–491.

[7] J. Blitzstein, P. Diaconis, A sequential importance sampling algorithm for generating random graphs with prescribed degrees, 2006.

[8] E. Bouillet, P. Dube, D. George, Z. Liu, D. Pendarakis, L. Zhang, Distributed multi-layered workload synthesis for testing stream processing systems, in: Winter Simulation Conference 2008 (WSC 2008), pp. 1003–1011.

[9] T. Britton, M. Deijfen, A. Martin-Loef, Generating simple random graphs with prescribed degree distribution, Journal of Statistical Physics 124 (2006) 1377–1397.

[10] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.M. Vincent, F. Wagner, Random graph generation for scheduling simulations, in: Proceedings

of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools '10), pp. 60:1–60:10.

[11] DIMACS Challenge, 10th DIMACS implementation challenge - Graph partitioning and graph clustering, 2012. http://www.cc.gatech.edu/dimacs10/.

[12] DOT Format, Dot tutorial and specification, 1988. http://www.graphviz.org/Documentation.php.

[13] D. Ediger, J. Riedy, H. Meyerhenke, D. Bader, Tracking structure of streaming social networks, in: IPDPS Workshops – 5th Workshop on Multithreaded Architectures and Applications (MTAAP'11), pp. 1691–1699.

[14] B. Gedik, H. Andrade, K.L. Wu, P.S. Yu, M. Doo, Spade: the System S declarative stream processing engine, in: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2008, pp. 1123–1134.

[15] M. Harchol-Balter, A.B. Downey, Exploiting process lifetime distributions for dynamic load balancing, ACM Transactions on Computer Systems 15 (1997) 253–285.

[16] R.M. Karp, Reducibility among combinatorial problems, in: Proceedings of Complexity of Computer Computations Symposium, pp. 85–103.

[17] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1999) 359–392.

[18] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, S. Rixner, Load distributing for locally distributed systems, IEEE Micro 21 (2001) 35–46.

[19] M. Lassnig, T. Fahringer, V. Garonne, A. Molfetas, M. Branco, Identification, modelling and prediction of non-periodic bursts in workloads, in:

2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 485–494.

[20] M. Mansour, M. Wolf, K. Schwan, Streamgen: A workload generation tool for distributed information flow applications, in: ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing, IEEE Computer Society, Washington, DC, USA, 2004, pp. 55–62.

[21] N. Parikh, Generating Random Graphs with Tunable Clustering Coefficient, Master's thesis, Virginia Polytechnic Institute and State University, 2011.

[22] S. Ray, R. Ungrangsi, F.D. Pellegrinin, A. Trachtenberg, D. Starobinski, Robust location detection in emergency sensor networks, in: The 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003).

[23] S. Rixner, Stream processor architecture, Kluwer Academic Publishers, 2002.

[24] W. Thies, Language and Compiler Support for Stream Programs, Ph.D. thesis, Massachusetts Institute of Technology, 2009.

[25] W. Thies, S.P. Amarasinghe, An empirical characterization of stream programs and its implications for language and compiler design, in: 19th International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 365–376.

[26] E. Veloso, V. Almeida, W.J. Meira, A. Bestavros, S. Jin, A hierarchical characterization of a live streaming media workload, IEEE/ACM Transactions on Networking 14 (2006) 133–146.

[27] F. Viger, M. Latapy, Efficient and simple generation of random simple connected graphs with prescribed degree sequence, in: Computing and Combinatorics Conference 2005 (COCOON '05).

[28] X.J. Zhang, S. Parekh, B. Gedik, H. Andrade, K.L. Wu, Workload characterization for operator-based distributed stream processing applications, in: DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pp. 235–247.