A Stream Algebra for Performance Optimization of Large Scale Computer Vision Pipelines

Mohamed A. Helala, Faisal Z. Qureshi, Ken Q. Pu
School of Faculty of Science, University of Ontario Institute of Technology, Oshawa, ON, Canada
{Mohamed.Helala, Ken.Pu, Faisal.Qureshi}@uoit.ca

There is a large growth in hardware and software systems capable of producing vast amounts of image and video data. These systems are rich sources of continuous image and video streams. This motivates researchers to build scalable computer vision systems that utilize data-streaming concepts for processing of visual data streams. However, several challenges exist in building large-scale computer vision systems. For example, computer vision algorithms have different accuracy and speed profiles depending on the content, type, and speed of incoming data. Also, it is not clear how to adaptively tune these algorithms in large-scale systems. These challenges exist because we lack formal frameworks for building and optimizing large-scale visual processing. This paper presents formal methods and algorithms that aim to overcome these challenges and improve building and optimizing large-scale computer vision systems. We describe a formal algebra framework for the mathematical description of computer vision pipelines for processing image and video streams. The algebra naturally describes feedback control and provides a formal and abstract method for optimizing computer vision pipelines. We then show that a general optimizer can be used with the feedback-control mechanisms of our stream algebra to provide a common online parameter optimization method for computer vision pipelines.

***Index Terms*—Stream Algebra, Online Vision Algorithms, Stream Processing, Large Scale Computer Vision Systems, Parameter Tuning, Performance Optimization.**

I. Introduction

Nowadays, our ability to generate images and videos has dramatically increased with the ubiquitous access to camera-equipped devices, such as smartphones and tablets. This created an increasing trend in recording personal memories and visually sharing stories, especially in social network platforms, such as YouTube [1] and Instagram. [2] We can anticipate that this trend will continue to increase in the future.

This huge growth of visual content motivates researchers to build systems that support web-scale computer vision by building scalable algorithms. The target is to process and analyze a vast amount of visual content as long-lasting streams. Such streams include image and video sequences (possibly infinite) from traffic-camera networks, vehicular vision systems, and vision-based Internet-of-things systems [1], [2], [3], [4]. We refer to this family of data streams as vision streams.

Moreover, many classical vision problems, such as tracking, object detection, object counting, edge detection, background modelling, etc., can be cast within the stream-processing framework. Online vision algorithms can be implemented as stream-processing functions. There are also several recent algorithms [5], [6], [7], [4], [8], [9], [10] that applied data-streaming concepts to efficiently solve different computer vision problems. Examples of online algorithms that deal with streams of images and video include streaming hierarchical video segmentation [7], photo stream alignment [9], human body segmentation [8], storyline inference from web photo streams [10], human activity prediction from video streams [5], and analysis of traffic-video streams [11].

Managing and processing data streams has been an active topic in the field of database and information systems [12], [13], [14], [15], [16]. However, most data-stream research has been focused on streams of textual, numerical, and semi-structured data. Consequently, the resulting stream-processing systems have been built and optimized for text-stream processing by relying on and extending database operators (such as relational algebraic operators and related analytical methods). To efficiently process textual data streams, the database community developed stream-algebra frameworks [17], [18], [19], [20]. Stream algebra extends relational algebras for data-stream processing and provides a formal language for mathematically defining workflow graphs. It defines a set of abstract and concurrent operators that take data streams as their operands to produce output streams. The operators have formal semantics to declare and construct streaming workflows as mathematical expressions. These algebras bring several advantages, such as providing a formal and abstract method for data-stream manipulation, resolving blocking operations, scheduling asynchronous and bulk-wise processing tasks, implementing dynamic execution plans, applying incremental evaluation, scaling up data processing, and defining common pipeline optimization and cost models.

Several platforms have been developed by building upon and extending stream algebras. Examples of such platforms include Apache Storm [21], Amazon Kinesis [22], Apache Kafka [23], and Spark Streaming [24]. The success of these platforms comes from their ability to 'intelligently' process vast amounts of streams and adapt to changes in computational resources. These platforms process data streams in an online fashion. Apache Storm [21] is an example of a distributed computation system for processing large-scale data streams. It is fault-tolerant and scalable with a guarantee of data processing. Storm [21] defines a pipeline as a directed graph topology with Spout operators and Bolt operators. A Spout reads data from a given source and submits them to Bolts. A Bolt is a generic data-processing operator that receives multiple input streams, processes them, and submits multiple output streams. The Amazon Kinesis [22] platform can acquire and process data streams in real time from thousands of different web sources at a data rate of several terabytes per hour. Apache Kafka [23] is a distributed messaging system that can read and

Manuscript received December 1, 2012; revised August 26, 2015. Corresponding author: M. Shell (email: http://www.michaelshell.org/contact.html).

[1] YouTube: https://www.youtube.com/ (*last accessed on 7 September 2017*).

[2] Instagram: https://www.instagram.com/ (*last accessed on 7 September 2017*).

write streams. Spark Streaming [24] allows scalable and fault-tolerant real-time stream processing. It receives data streams and splits them into batches that are processed in parallel using a set of stream-processing operators, extending the relational algebra.

These large-scale processing platforms provide different implementations of the database stream-algebra frameworks and are optimized for processing textual streams, however, researchers in computer vision have tried to utilize these platforms for processing vision streams. For example, Yu et al. [25] developed a video parsing and evaluation platform using Spark Streaming and Kafka for processing long surveillance videos. In this platform, users write computer vision processing modules using Spark Streaming, and Kafka is used for communicating messages between different modules. Tabernik et al. [26] developed a web service for object detection using hierarchical models. Their method is implemented as a pipeline in Storm using custom-built Spouts and Bolts. Zhang et al. [27] developed a system architecture for online surveillance-video processing based on Kafka and Spark streaming. In this system, users can write video tasks that process blocks of video data.

Despite these efforts to customize text-stream-processing platforms for computer vision, there are still several problems in building large-scale computer vision systems. These problems stem from the nature of image and video streams and the complexity of computer vision algorithms. Vision streams are more complex than text streams. They usually have high dimensional features, complex/multi-modal data, and a wide range of noisy samples. This induces a very large number of intermediate results generated by computer vision tasks in large-scale systems, thus creating large latencies in moving data in a distributed system. Moreover, computer vision tasks deal with different types of videos and images, such as binary images, images with different colour models, and videos with different frame rates, environments, illuminations, etc. To deal with these different conditions, a computer vision task can be implemented using different algorithms with different accuracy and speed profiles depending on the content, type, and speed of incoming data. Thus, a large-scale computer vision system should enable dynamic reconfiguration of the system to switch between different runtime profiles to match changes in incoming streams. In addition, computer vision algorithms usually require a larger number of input parameters than text-stream-processing algorithms. It is not clear how to adaptively tune this large number of parameters in large-scale systems to dynamically adapt to changes in content, type, and speed of incoming vision streams. These problems suggest that a formal framework is needed to solve the challenges of large-scale processing of vision streams.

This paper presents a formal and scalable framework for building and optimizing large-scale visual processing. We develop a stream-algebra framework for manipulating vision streams, which can be used to mathematically describe the pipelines of several state-of-the-art techniques in computer vision. The algebra has a common notation and defines a set of concurrent algebraic operators that provide a new abstraction for computer vision operations and can be used to build scalable computer vision pipelines. It also provides a natural description of feedback control, which is the fundamental task for several advanced optimizations, such as adaptive optimization and parameter tuning. We show how the algebra can describe two online computer vision algorithms for traffic video analysis [28] and iterative optimization [10]. Moreover, We use the feedback primitives of the stream algebra for developing common optimization methods for parameter tuning of large-scale pipelines. We show that the algebra can effectively implement and scale a general sequential model-based optimization method [29] for parameter tuning in large-scale computer vision pipelines. For simplicity and without loss of generality, we use the online road boundary detection method of [28] as our case study, and show the effectiveness of our stream algebra in implementing common optimization methods for parameter tuning in large scale vision stream processing.

II. Related Work

Stream Algebra. Building formal stream-algebra frameworks has been addressed in the database community [17], [18], [19], [20] for efficiently processing textual data streams. A stream algebra is a formal language for mathematically defining workflow graphs. It defines a set of abstract and concurrent operators that take data streams as their operands to produce output streams. The operators have formal semantics to declare and construct streaming workflows as mathematical expressions. Stream algebras exist in the database literature to provide an effective description of queries on event or relational streams. They provide a framework for query analysis and optimization. For example, Broy et al. [18] defined streaming pipelines as data-flow networks. The core of the algebra is a set of algebraic operators that can construct data-flow networks as graphs of stream processing functions. Carlson and Lisper [19] presented an event detection algebra for reactive systems in which the system should respond to external events and produce corresponding actions. They provided an imperative algorithm to evaluate the algebraic expressions and produce the corresponding output streams. Demers et al. [20] proposed another stream algebra that extends relational algebra to describe queries on event streams. Their algebraic extensions include a set of stream operators with well defined semantics for processing stock streams.

Building Efficient Online Algorithms. There is currently progress in developing online computer vision pipelines [5], [6], [7], [4], [8], [9], [10] that manipulate vision streams. These applications span several areas, such as analysis of web photo collections, activity recognition, surveillance cameras, and satellite imagery. For example, Schuster et al. [30] proposed a method for real-time detection of unusual regions in surveillance-video streams. The method partitions each image and extracts a local model for each partition. Then, the model is continuously updated toward scene changes by applying several heuristic rules. The method was applied to guide camera operators to areas of attention.

Cao et al. [31] proposed a method for recognizing human activities from video streams in which part of the activities are missing. They cast the problem as a probabilistic framework and used sparse coding to calculate the likelihood of a test video belonging to a certain activity. Kim et al. [10] proposed an algorithm to build common storylines from Flickr photo streams and to discover the relations between them. Their method jointly aligns and segments large-scale web photo streams by applying message-passing-based optimization. Xuand et al. [7] also proposed a recent streaming framework that approximates full video hierarchical segmentation

to work in a constant memory space. Their algorithm works in a feedforward streaming fashion by dividing an input video stream into a set of clips. Then, for each clip, a segmentation hierarchy is generated using an automatic semi-supervised method that uses a Markovian assumption to relate the current hierarchy to the previous one.

Despite the previous progress in developing online vision algorithms, there are also challenges resulting from user-defined vision algorithms. For example, the algorithms can have unpredictable or high processing rates based on the input size and content, which causes some algorithms to employ different approximations to support real-time requirements. Many algorithms also have several runtime parameters, which are selected by isolated experimental evaluations, and usually lack runtime tuning when integrated into other algorithms. Another important challenge is the need to design vision algorithms for heterogeneous hardware platforms (mobile to heterogeneous hardware processors), which makes the algorithms hard to debug and extend.

Feedback Control in Data-Stream Systems. Feedback control is a very important area in the field of control systems and has a wide range of applications in several industrial plants. Feedback-control systems can have one or more feedback loops. Such systems have controlled processes that are univariate or multivariate [32]. Univariate processes have a single controlled input variable (or parameter) and produce a single output variable. Thus, only one feedback loop with a single controller is required to self-adjust the output variable. The resulting control system in this case is referred to as a *single-loop feedback-control system*. On the other hand, multivariate processes are typically found more often in practice, where a process has multiple controlled input variables and produces one or more output variables. Moreover, in more complex scenarios, a multivariate process may have feedback signals coming from the outputs of other processes. To handle such cases, the control system must have multiple feedback loops and is referred to in literature as a *multi-loop feedback-control system*.

Parameter Tuning. Most computer vision algorithms rely on parameters to control runtime performance or output accuracy. To find the optimal parameter settings, we need to perform parameter tuning. This can be performed either manually, semi-automatically, [33] or automatically [34], [35], [36]. For a vision pipeline that contains several chained stream operators, parameter tuning becomes more challenging. This is because each operator can implement a vision algorithm that has a different set of parameters. Thus, the parameter space becomes large. In addition, these parameters need to be continuously updated toward changes in the input vision stream.

Several parameter-tuning algorithms have been proposed in computer vision for solving specific problems. For example, Sherrah [34] proposed an algorithm for continuous real-time parameter tuning of a people-tracking surveillance system. This algorithm works in two phases, an offline learning phase and an online tuning phase. In the offline learning phase, the algorithm learns the best set of parameters. During the online phase, the algorithm incrementally adapts the parameters in a continuous fashion to react to changes in input data. Parameter tuning for long-term tracking was explored by Supancic et al. [35], who proposed an online tracking algorithm that uses self-paced learning to continuously adapt the parameters of an appearance model to the tracked object. Chau et al. [36] also studied parameter tuning for tracking algorithms. They proposed an online technique to tune the parameters of a tracking algorithm to different scene contexts. This technique uses an offline learning stage to learn the tracker parameters in different contexts. Then, an online stage continuously examines the tracking quality. If the quality is not good enough, the algorithm detects the current context and tunes the tracking parameters using the previously learned values.

III. Stream Algebra

Our stream algebra [37], [11], [38] consists of three main components: a common notation for expressing pipelines, a set of data-processing and flow-control operators, and the formal semantics used to write algebraic expressions. This section gives an overview of the notation and semantics of the algebra and its operators.

A. Notation

We refer to the set of all streams as $\mathbf{S}$. To signify that a stream contains tuples of a specific type T, we use the notation $\mathbf{S}\langle T\rangle$.

Definition 3.1 (Stream operator): A stream operator is a function $h : \mathbf{S}^m \rightarrow \mathbf{S}^n : S^1_{\text{in}}, \ldots, S^m_{\text{in}} \rightarrow S^1_{\text{out}}, \ldots, S^n_{\text{out}}$ that maps m input streams to n output streams. The operator performs the mapping $\langle x, y\rangle \in h$ to transform each $x = (x^1, .., x^m)$ to $y = (y^1, .., y^n)$, where $x \in S^1_{\text{in}} \times ... \times S^m_{\text{in}}$, and $y \in S^1_{\text{out}} \times ... \times S^n_{\text{out}}$. Also for $1 \leq i \leq m$ and $1 \leq j \leq n$, the operator has the consumption data rate d^i_{in} for S^i_{in} and the production data rate d^j_{out} for S^j_{out}.

The following constructs are used to define the algebra operators:

- Atomicity: We define a set of statements executed as an atomic operation using { *statements* }.
- Shared state: A state defined as **state** u indicates a shared state and can be accessed by the next loops.
- Concurrency: An infinite loop is defined as **loop** : *body of loop*. The loop applies the body logic iteratively on input stream tuples. If an operator defines several concurrent loops, they all share the defined states, and $\textbf{loop}_j$ is designated as the j-th loop.
- Stream I/O: The function $x \leftarrow s$ reads a tuple from stream s into x, and the function $e \rightarrow s$ writes a tuple e to stream s.
- Attribute access: We use $x.y$ to access attribute y from the vector $x = (w, z, y)$.

Definition 3.2 (Operator declaration): A stream operator h is a mapping function that can have zero or more parameters. The parameters are the user-defined functions and their functional parameters. Given the functional signature of each parameter, we derive a stream operator and declare it using the following format:

$$\frac{f_1 : \text{signature}_1, \ldots, f_k : \text{signature}_k}{h(f_1, ..., f_k) : \mathbf{S}^m \rightarrow \mathbf{S}^n},$$

where signature_i defines the mapping performed by the function f_i. Moreover, S_{in} is synchronized with S_{out}, $d_{\text{out}} \leq d_{\text{in}}$, if data consumption from S_{in} and production to S_{out} happen in the same loop. They are asynchronous, d_{out} decoupled

from d_{in}, if data consumption and production happen in different loops. Consequently, a synchronous operator produces synchronous output streams, and an asynchronous operator produces at least one asynchronous output stream.

Our stream algebra defines three classes of operators: 1) main operators which are first-order primitive operators that process a predefined number of streams; 2) additional operators which are higher-order primitive operators that extend the algebra to process collections of streams; and 3) auxiliary operators which are useful non-primitive operators that are defined in terms of other main and additional operators.

B. First-order Operators

We start by formulating the main operators that typically process a single-input stream to produce a predefined number of output streams.

*Definition 3.3 (***Map):** the operator $\text{MAP}(f, p_0) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle Y\rangle$ is parameterized by a list of user-defined mapping functions $f = (f_1, f_2, ..., f_n)$ and a list of parameters $p_0 = (p_1, p_2, ..., p_n)$. The operator performs the mapping $\langle x, y\rangle \in \text{MAP}(f, p_0)$, and $y = f_i(x, p_i)$, where $f_i : X \times P \rightarrow Y$ has its behaviour controlled by the parameters $p_i : P$. The operator has two state variables, the function index i and the list of parameters p with initial value $p = p_0$. We have $X : H \times D$, where each $x \in X$ is a vector containing a header section of type H, and a data section of type D. The header section may include a set of n parameters, where $P^n \subset H$. Also $X : D$ if $H = \varnothing$. A lookup function updates the state variables by reading the header section of each incoming tuple x. The operator has $d_{\text{out}} = d_{\text{in}}$:

$$\frac{f : \text{LIST}\langle X \times P \rightarrow Y\rangle}{\text{MAP}(f, p_0) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle Y\rangle}$$

state $i = 0;\ p = p_0$
loop : $x \leftarrow S_{\text{in}}$
$j, y = \text{lookup}(x)$
if j is defined **then** $i = j$
if $y \neq null$ **then** $p[i] = y$
$f[i](x, p[i]) \rightarrow S_{\text{out}}$

*Definition 3.4 (***Reduce):** This operator is similar to Map, but it keeps track of an additional internal shared state $u : U$. The operator is parameterized by a list of mapping functions $g : \text{LIST}\langle U \times X \times P \rightarrow U \times Y\rangle$ and an initial state u_0:

$$\frac{u_0 : U,\ \ g : \text{LIST}\langle U \times X \times P \rightarrow U \times Y\rangle}{\text{REDUCE}(g, u_0, p_0) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle Y\rangle}$$

state $u = u_0;\ i = 0;\ p = p_0$
loop : $x \leftarrow S_{\text{in}}$
$j, y = \text{lookup}(x)$
if j is defined **then** $i = j$
if $y \neq null$ **then** $p[i] = y$
$u, z = g[i](u, x, p[i])$
$z \rightarrow S_{\text{out}}$

For a Map operator, if $p = p_0 = \varnothing$, then we can omit the set of parameters and write Map (f), where $f : \text{LIST}\langle X \rightarrow Y\rangle$. Moreover, if $f = \{f_1\}$ with only one function $h = f_1$, we can simplify expressions by writing Map (h). Similarly, these assumptions apply to Reduce.

*Definition 3.5 (***Filter):** the operator $\text{FILTER}(\theta) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle X\rangle \times \mathbf{S}\langle X\rangle$ is parameterized by a predicate $\theta : X \rightarrow$ boolean. This operator can be also referred to as Partition or Split. It performs the mapping $\langle x, x\rangle \in S_{\text{in}} \rightarrow S^1_{\text{out}}$ if $\theta(x) = \texttt{True}$, and $\langle x, x\rangle \in S_{\text{in}} \rightarrow S^2_{\text{out}}$ if $\theta(x) = \texttt{False}$. The operator has $\forall i \in \{1, 2\} : d^i_{\text{out}} \leq d_{\text{in}}$:

$$\frac{\theta : X \rightarrow \text{Boolean}}{\text{FILTER}(\theta) : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle X\rangle \times \mathbf{S}\langle X\rangle}$$

loop : $x \leftarrow S_{\text{in}}$
if $\theta(x)$ **then** $x \rightarrow S^1_{\text{out}}$ **else** $x \rightarrow S^2_{\text{out}}$

*Definition 3.6 (***Source):** The operator $\text{SOURCE}(u_0, h) : \varnothing \rightarrow \mathbf{S}\langle Y\rangle$ generates S_{out} at the data rate d_{out}, and is parameterized by an initial shared state $u_0 : U$ and a generator function $h : U \rightarrow U \times Y$:

state $u = u_0$
loop : $u, y = h(u)$
$y \rightarrow S_{\text{out}}$

*Definition 3.7 (***Copy):** The operator $\text{COPY}(n) : \mathbf{S} \rightarrow \mathbf{S}^n$ performs the mapping $\left\langle x, (x \overset{\times n}{\cdots})\right\rangle \in \text{COPY}(n)$ by duplicating every input tuple x n-times to n outgoing streams. The operator has $\forall i \in \{1, ..., n\} : d^i_{\text{out}} = d_{\text{in}}$:

loop : $x \leftarrow S_{\text{in}}$
$x \rightarrow S_{\text{out}}[i]$ for all $i \leq n$

*Definition 3.8 (***Ground):** The operator $\text{GROUND} : \mathbf{S} \rightarrow \varnothing$ destroys the incoming stream:

loop : $\leftarrow S_{\text{in}}$

C. Rate-control Operators

The main operators discussed in the previous section are synchronous operators. In this section, we present a set of main asynchronous operators that have one or more asynchronous output streams. Given an asynchronous operator $h : S_{\text{in}} \rightarrow S_{\text{out}}$ with $k = d_{\text{out}}/d_{\text{in}}$, we define the asynchronous mapping $\langle x, y\rangle_{\times k} \subset h$ based on the value of k. If $k \geq 1$, d_{out} is faster than or equal to d_{in}, then $\langle x_i, y_i\rangle_{\times k}$ maps $S_{\text{in}} = \{x_i | i \geq 0\}$ to $S_{\text{out}} = \{y_i \overset{\times k}{\cdots} | i \geq 0\}$. In this case, $\langle x_i, y_i\rangle_{\times k} = \{\langle x_i, y_i\rangle \overset{\times k}{\cdots}\}$ performs the mapping $\langle x_i, y_i\rangle$ k-times on each input tuple $x_i \in S_{\text{in}}$ to produce the k output tuples $\{y_i \overset{\times k}{\cdots}\}$. If $k < 1$, d_{out} is slower than d_{in}, then the operator h samples S_{in} using the sampling rate $k = 1/s$ to generate the samples $\{x_i | i = js, j \geq 0\}$, while discarding other input tuples with $i \neq js$. The mapping $\langle x_i, y_i\rangle_{\times k} = \{\langle x_i, y_i\rangle | i = js\}$ with $\langle x_i, y_i\rangle \in h$ is then performed only once on each sample x_i to produce the stream $S_{\text{out}} = \{y_i | i = js, j \geq 0\}$.

*Definition 3.9 (***Latch):** The operator $\text{LATCH} : \mathbf{S} \rightarrow \mathbf{S} \times \mathbf{S}$ receives the input stream S_{in} and produces two output streams, an asynchronous output S^2_{out} and a synchronous output S^1_{out}. The operator performs the synchronous mapping $\langle x, x\rangle \in S_{\text{in}} \rightarrow S^1_{\text{out}}$ with $d^1_{\text{out}} = d_{\text{in}}$, and the asynchronous mapping $\langle x, x\rangle_{\times k} \subset S_{\text{in}} \rightarrow S^2_{\text{out}}$ with $k = d^2_{\text{out}}/d_{\text{in}}$. Notice that the asynchronous mapping produce k duplicates for every input x if $d^2_{\text{out}} > d_{\text{in}}$:

$$\textbf{loop}: \quad x \leftarrow S_{\text{in}} \quad \{u = x; x \rightarrow S^1_{\text{out}}\} \qquad \textbf{loop}: \quad \{u \rightarrow S^2_{\text{out}}\}$$

*Definition 3.10 (***Cut):** The operator $\text{CUT}() : \mathbf{S} \rightarrow \mathbf{S} \times \mathbf{S}$ is similar to Latch; however, the operator performs the asynchronous mappings $\{\langle x, x\rangle\} \bigcup \langle x, nil\rangle_{\times k-1}$ when $k \geq 1$. This guarantees that each input $x \in S_{\text{in}}$ is asynchronously written only once to S^2_{out}. If S^2_{out} has a faster data rate than S_{in}, then nil is used for the extra writes:

$$\textbf{state} \quad u = \textbf{nil}$$
$$\textbf{loop}: \quad x \leftarrow S_{\text{in}} \quad \{u = x; x \rightarrow S^1_{\text{out}}\} \qquad \textbf{loop}: \quad \{y = u; u = \textbf{nil}\} \quad y \rightarrow S^2_{\text{out}}$$

We also define the auxiliary operator $\text{CUT}^G : \mathbf{S} \rightarrow \mathbf{S}$ as:

$$S, S_{\text{out}} = \text{CUT}(S_{\text{in}}); \text{GROUND}(S).$$

LATCH^G is similarly defined by replacing Cut with Latch.

D. Higher-order Operators

Section III-B describes the first-order operators, where the number of input and output streams are predefined by the operator definition. The operators also are parameterized by simple functions. Any pipelined composition of first-order operators results in first-order operators as well. This section extends the algebra by presenting additional higher-order operators. These operators process collections of streams $\mathcal{S} = \{S^1, ..., S^n\}$, and have functions and first-order operators as their input parameters.

*Definition 3.11 (***Mult):** The operator $\text{MULT}() : \mathbf{S}^n \langle X\rangle \rightarrow \mathbf{S}\langle \text{LIST}\langle X\rangle\rangle$ has n incoming streams, and one output stream S_{out}. The operator performs the synchronous mapping $\langle x, y\rangle \in \text{MULT}()$, where every set of input values $x = (x^1, ..., , x^n)$, $x \in S^1_{\text{in}} \times ... \times S^n_{\text{in}}$, is written to the output list $y = \{x^1, ..., x^n\}$, and $y \in S_{\text{out}}$. The operator has $d_{\text{out}} = \min_{0 \leq i \leq n} d^i_{\text{in}}$:

$$\textbf{loop} \quad : \quad \begin{bmatrix} \leftarrow \mathcal{S}_{\text{in}}[1] \\ ... \\ \leftarrow \mathcal{S}_{\text{in}}[n] \end{bmatrix} \rightarrow S_{\text{out}}$$

*Definition 3.12 (***Add):** The operator $\text{ADD}() : \mathbf{S}^n \rightarrow \mathbf{S}$ also has n incoming streams and one output stream. The operator performs the mappings $\forall i \in \{1, ..., n\} : \langle x^i, x^i\rangle \in S^i_{\text{in}} \rightarrow S_{\text{out}}$. The operator asynchronously reads values from the input streams and performs the best effort to sequentially write them to the output stream. The tuples $x^i_t \in S^i_{\text{in}}$ and $x^j_{t'} \in S^j_{\text{in}}$, received at the times t and t', have the order $\{x^i_t, x^j_{t'}\} \subset S_{\text{out}}$ if $t < t'$. The order is non-deterministic if $t = t'$. Here $d_{\text{out}} = \sum_{i \in [1,n]} d^i_{\text{in}}$:

$$\text{for all } i \leq \textbf{len}(\mathcal{S}_{\text{in}}) \quad \textbf{loop}: \{x \leftarrow \mathcal{S}_{\text{in}}[i]; x \rightarrow S_{\text{out}}\}$$

*Definition 3.13 (***Scatter):** The operator $\text{SCATTER}(f, p) : \mathbf{S} \rightarrow \mathbf{S}^n$ receives one input stream and generates n output streams. The operator is parameterized by two functions: $f : X \rightarrow \text{LIST}\langle Y\rangle$ and $p : Y \times \mathbb{N} \rightarrow \mathbb{N}$, and performs the mapping $\forall j \in [1, n], i \in [1, m] : \langle x, y^i\rangle \in S_{\text{in}} \rightarrow S^j_{\text{out}}$ for $y = f(x) = (y^1, ..., y^m)$, $j = p(y^i, i)$, and $m \geq n$. Here f is a generator function that computes output values, and p is a partition function that maps each output value y^i to the $p(y^i, i)$-th output stream. The operator is synchronous, where $\forall i \in \{1, ..., n\} : d^i_{\text{out}} = d_{\text{in}}$:

$$\frac{f : X \rightarrow \text{LIST}\langle Y\rangle \quad , \quad p : Y \times \mathbb{N} \rightarrow \mathbb{N}}{\text{SCATTER}(f, p) : \mathbf{S}\langle X\rangle \rightarrow \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle}$$

$\text{SCATTER}(f, p) : S_{\text{in}} \mapsto \mathcal{S}_{\text{out}}$
let $\mathcal{S}_{\text{out}} = \text{EMPTY-LIST}\langle \mathbf{S}\langle X\rangle\rangle$
loop: $y = f(\leftarrow S_{\text{in}})$
$y[i] \rightarrow \mathcal{S}_{\text{out}}[p(y[i], i)]$ for all $i \leq \textbf{len}(y)$

*Definition 3.14 (***Merge):** The operator $\text{MERGE}(f) : \mathbf{S}^n \rightarrow \mathbf{S}$ is the "inverse" of Scatter in the sense that it merges a collection of n incoming streams back into a single outgoing stream. The operator reads from n incoming streams into a buffer $y = \{y^1, ..., y^n\}$, where $\forall i \in \{1, ..., n\} : y^i =\leftarrow S^i_{\text{in}}$ if and only if y^i is nil. When nil $\notin y$, the operator performs the mapping $\left\langle y, y^{i^*}\right\rangle \in S^1_{\text{in}}, ..., S^n_{\text{in}} \rightarrow S_{\text{out}}$, and sets $y^{i^*} =$ nil, where $i^* = \text{argmin}_{\leq}\{f(y^i)\}$. The function $f : X \rightarrow (Y, \leq)$ is a selection function that has a partial order over Y, and is used to determine the smallest element (w.r.t. $\leq$) in y. The operator has $d_{\text{out}} = \min_{0 \leq i \leq n} d^i_{\text{in}}$:

$$\frac{f : X \rightarrow (Y, \leq)}{\text{MERGE}(f, \leq) : \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle \rightarrow \mathbf{S}\langle X\rangle}$$

$\text{MERGE}(f) : \mathcal{S}_{\text{in}} \mapsto S_{\text{out}}$
State: y where $|y| = |\mathcal{S}_{\text{in}}|$.
for each $S_{\text{in}} = \mathcal{S}_{\text{in}}[i]$:
loop: {**if** $y[i] ==$ **nil then** $y[i] \leftarrow S_{\text{in}}$}
end for
loop: **if nil** $\notin y$ **then**
$i^* = \text{argmin}_{\leq}\{f(y[i])\}$
$\{y[i^*] \rightarrow S_{\text{out}}; y[i^*] =$ **nil**$\}$
end if

We also define the following auxiliary higher-order operators which we found useful in algebraic expressions:

*Definition 3.15 (***Left-Mult):** The operator $\text{LEFT-MULT}(k = 1) : \mathbf{S}^{m+k}\langle X\rangle \rightarrow \mathbf{S}\langle \text{LIST}\langle X\rangle\rangle$ is an auxiliary operator and is similar to Mult; however, it applies a Latch^G on each stream $S \in \{S^i_{\text{in}} | i = m + 1, .., m + k\}$ of the last k input streams, and $k = 1$ by default. This produces the k asynchronous streams $\{S^j | j = 1, .., k\}$. The operator then applies Mult on the streams $\{S^1_{\text{in}}, ..., S^m_{\text{in}}, S^1, ..., S^k\}$. The operator performs the mapping $\langle (x, z), y\rangle \in \mathbf{S}^m \times \mathbf{S}^k \rightarrow \mathbf{S}$, where $y = \{x^1, ...x^m, w^1, ..w^n\}$, $x \in S^1_{\text{in}} \times ... \times S^m_{\text{in}}$, $z \in S^{m+1}_{\text{in}} \times ... \times S^{m+k}_{\text{in}}$, and for $j \in \{1, .., k\}$: $\left\langle z^j, w^j\right\rangle \in S^{m+j}_{\text{in}} \rightarrow S^j$:

let $\mathcal{S} = \text{EMPTY-LIST}\langle \mathbf{S}\langle X\rangle\rangle$
for $1 \leq i \leq k$, and $|\mathcal{S}| = m + k$:
$\mathcal{S}[i] \triangleq \text{LATCH}^G(\mathcal{S}_{\text{in}}[m + i])$
$S_{\text{out}} \triangleq \text{MULT}(\mathcal{S}_{\text{in}}[1], ..., \mathcal{S}_{\text{in}}[m], \mathcal{S}[1], ..., \mathcal{S}[k])$

RightMult can be similarly defined.

*Definition 3.16 (***List-Map):** The operator $\text{LIST-MAP}(h) : \mathbf{S}^n \rightarrow \mathbf{S}^n$ is a higher-order auxiliary operator that has collections of streams as input and output. The operator is parameterized by a composition of first-order operators $h : \mathbf{S} \rightarrow \mathbf{S}$, and performs the mappings $\forall i \leq n : S^i_{\text{out}} = h(S^i_{\text{in}})$:

$$\frac{h : \mathbf{S}\langle X\rangle \rightarrow \mathbf{S}\langle Y\rangle}{\text{LIST-MAP}(h) : \text{LIST}\langle \mathbf{S}\langle X\rangle\rangle \rightarrow \text{LIST}\langle \mathbf{S}\langle Y\rangle\rangle}$$

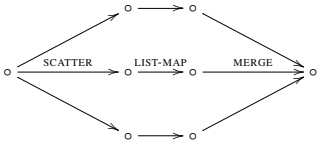

Fig. 1: An example of a concurrency pattern expressed in our algebra.

$\text{LIST-MAP}(h) : \mathcal{S}_{\text{in}} \mapsto \mathcal{S}_{\text{out}}$
let $\mathcal{S}_{\text{out}} = \text{EMPTY-LIST}\,\langle \mathbf{S}\,\langle Y\rangle\rangle$
for all $i \leq \textbf{len}(\mathcal{S}_{\text{in}})$
loop : $y \leftarrow h(\mathcal{S}_{\text{in}}[i])$
$y \rightarrow \mathcal{S}_{\text{out}}[i]$

The higher-order operators allow the algebraic description of large-scale stream pipelines processing collections of streams and having high degrees of concurrency. Figure 1 shows the Scatter, List-Map, and Merge parallel processing pattern naturally expressed in our algebra. This pattern is popularly utilized in several concurrent processing environments, such as distributed clusters, graphics processing units (GPUs), and multicore processing [39], [40].

IV. Stream Algebra to Basic Network Algebra

In this section, we provide a theoretical study of our stream algebra and base its semantics on the equational theory of basic network algebra (BNA) [41], [18], [42]. BNA is part of the algebra of flownomials [18] that is common to different classes of networks including flowcharts and data-flow networks. BNA defines a set of compositional and constant operators that have a set of correct and complete axioms for flowcharts/networks. Additional constant operators are required for describing branching connections, and are defined based on the class of network. In this work, we build upon the data transformer model of BNA presented by [41], [42] for synchronous and asynchronous data-flow networks. This model describes a data-flow network as a set of parallel data transformation functions connected using communication channels, where a function $f : \mathbf{S}^m \rightarrow \mathbf{S}^n$ maps m input data streams to n output data streams. Notice that a data transformation function in this model is a stream operator in our stream algebra by Definition 3.1.

A. Main BNA Operators

We start by describing the main BNA compositional and constant operators defined by the data transformer model of BNA [41], [42] for both synchronous and asynchronous data-flow networks. Given the two stream operators $h : \mathbf{S}^m \rightarrow \mathbf{S}^n$, and $h' : \mathbf{S}^p \rightarrow \mathbf{S}^q$, the main BNA operators are defined as follows:

*Definition 4.1 (***Identity):** the constant operator $\mathsf{I}_n : \mathbf{S}^n \rightarrow \mathbf{S}^n$ forwards the input to output, where $\forall x \in S^1_{\text{in}} \times ... \times S^n_{\text{in}}$, $\langle x, x\rangle \in \mathsf{I}_n$. This is the same as $\text{LIST-MAP}(h)$, where $h = \text{MAP}(\lambda\, x :\ x)$.

*Definition 4.2 (***Transposition):** the constant operator ${}^m\times^n : \mathbf{S}^m \times \mathbf{S}^n \rightarrow \mathbf{S}^n \times \mathbf{S}^m$ performs the mapping $\langle (x, y), (y, x)\rangle \in {}^m\times^n$, where $x = (x^1, .., x^m)$, and $y = (y^1, ..., y^n)$, for $x \in S^1_{\text{in}} \times ... \times S^m_{\text{in}}$, and $y \in S^{m+1}_{\text{in}} \times ... \times S^{m+n}_{\text{in}}$. Notice that we can write this operator in our stream algebra as ${}^m\times^n = \text{MULT}() \circ \text{SCATTER}(f, \lambda\ z, i :\ i)$, and the generator function f maps the incoming values $(x^1, .., x^m, y^1, ..., y^n)$ to outgoing values $(y^1, ..., y^n, x^1, .., x^m)$.

*Definition 4.3 (***Parallel):** the composition $h \mathbin{+\!\!+} h' : \mathbf{S}^m \times \mathbf{S}^p \rightarrow \mathbf{S}^n \times \mathbf{S}^q$ performs the mapping $\langle (x, z), (y, w)\rangle \in h \mathbin{+\!\!+} h'$, such that $\langle x, y\rangle \in h$ and $\langle z, w\rangle \in h'$. This composition indicates that the two operators h and h' process their inputs in parallel. Given $h = h_1 = h_2 ... = h_n$, we can write a ListMap operator in BNA as,

$$\text{LIST-MAP}(h) = \mathop{\mathbin{+\!\!+}}_{i=1}^{n} h_i = h_1 \mathbin{+\!\!+} h_2 \mathbin{+\!\!+} ... \mathbin{+\!\!+} h_n. \tag{1}$$

*Definition 4.4 (***Sequential):** the composition $h \circ h' : \mathbf{S}^m \rightarrow \mathbf{S}^q$ performs the mapping $\langle x, w\rangle \in h \circ h'$, where $n = p$ and $\exists z$ such that $\langle x, z\rangle \in h$ and $\langle z, w\rangle \in h'$. We can also define $\mathop{\circ}_{i=1}^{n} h_i = h_1 \circ h_2 \circ ... \circ h_n$.

*Definition 4.5 (***Feedback):** Given the operator $h : \mathbf{S}^m \times \mathbf{S}^p \rightarrow \mathbf{S}^n \times \mathbf{S}^p$, the feedback composition $h\uparrow^p$: $\mathbf{S}^m \rightarrow \mathbf{S}^n$ is defined iteratively as follows:

- First step: let (x_0, z_0) be the inputs to operator h at time $i = 0$, where $x_0 \in S^1_{\text{in}} \times ... \times S^m_{\text{in}}$, and $z_0 \in S^{m+1}_{\text{in}} \times ... \times S^{m+p}_{\text{in}}$.
- Iterative step: The set of iterative mappings $\{\langle (x_i, z_i), (y_i, z_{i+1})\rangle \in h | i \geq 0\}$ define $h\uparrow^p$ as the set of mappings $\{\langle x_i, y_i\rangle \in h\uparrow^p | i \geq 0\}$, and the operator $\uparrow^p$ feeds the last p output streams of h to its last p input streams, where $\forall j \in \{1, ..., p\} : S^{m+j}_{\text{in}} = S^{n+j}_{\text{out}}$.

We extend this feedback definition to feedback-control using our stream algebra. let us assume the feedforward pipeline $\mathop{\circ}_{i=1}^{k} h_i$ in Figure 2a, where each h_i is either a Map or Reduce operator. Each operator h_i has an input stream I_{j-1} and an output stream I_j. The streams I_0 and I_k define the pipeline input and output streams, respectively. Each streaming operator can be treated as a multivariate process with the user-defined parameters representing the input-controlled variables.

Figure 2b shows a multi-loop feedback-control system with two feedback loops. Each feedback loop has a return stream R_i for $i \in \{1, 2\}$, and a sequence of operators $f^i_1, ..., f^i_n$ that process R_i to generate the feedback stream F_i. This multi-loop feedback-control system can be described as:

$$\begin{aligned} h_1 \circ (&\text{LEFT-MULT}() \circ (\mathop{\circ}_{i=2}^{j-1} h_i) \circ \\ &(\text{LEFT-MULT}() \circ (\mathop{\circ}_{i=j}^{k} h_i) \circ \\ &\quad \text{COPY}(3) \circ (\mathsf{I}_1 \mathbin{+\!\!+} \mathop{\circ}_{l=1}^{n} f^1_l \mathbin{+\!\!+} \mathop{\circ}_{l=1}^{n} f^2_l) \\ &) \uparrow^1 \\ ) \uparrow^1 &. \end{aligned}$$

Here the Copy operator is applied as follows:

$$I'_k, R_1, R_2 \triangleq \text{COPY}(3)(I_k). \tag{2}$$

The streams $\{I'_k, R_1, R_2\}$ are forwarded to the parallel composition $(\mathsf{I}_1 \mathbin{+\!\!+} \mathop{\circ}_{l=1}^{n} f^1_l \mathbin{+\!\!+} \mathop{\circ}_{l=1}^{n} f^2_l)$. The stream I'_k is passed through I_1 to form the output stream. The streams R_1 and R_2 are processed by $\mathop{\circ}_{l=1}^{n} f^1_l$ and $\mathop{\circ}_{l=1}^{n} f^2_l$ to produce F_1 and F_2, respectively. The inner feedback operator $\uparrow^1$ forwards F_2 to the last input of the inner Left-Mult operator. Similarly, F_1 is forwarded to the last input of the outer Left-Mult operator. Here, Left-Mult is a merging operator that mimic the same functionality of a controller in traditional feedback-control systems.

This algebraic description shows that our stream algebra can naturally express feedback-control. For the task of parameter tuning, we can evaluate the output and generate a feedback stream that optimizes a set of controlled operators. If we

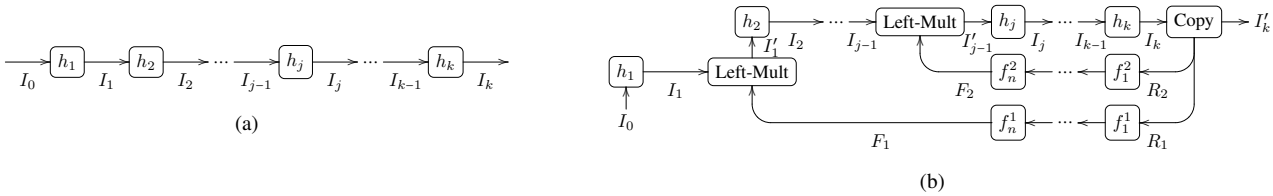

Fig. 2: Example of a multi-loop feedback-control system. a) Feedforward streaming pipeline. b) Multi-loop feedback-control system for controlling a set of operators in the streaming pipeline shown in (a).

replace Left-Mult with Add, the elements of every feedback stream will be interleaved with the corresponding input stream. Such interleaving is useful in iterative optimization, where the output is fed back to the input for reprocessing. Furthermore, the feedback loops in Figure 2b have the feedforward pipeline synchronized with the feedback loops. If we replaced Copy with Cut or Latch, the feedback loops will be asynchronous and independent of the feedforward pipeline.

TABLE I: The main axioms of BNA. The axioms are valid for our stream algebra. f, g, and h are stream operators.

B1	$f \mathbin{+\!\!+} (g \mathbin{+\!\!+} h) = (f \mathbin{+\!\!+} g) \mathbin{+\!\!+} h$	R1	$g \circ (f \uparrow^m) = ((g \mathbin{+\!\!+} \mathsf{I}_m) \circ f) \uparrow^m$
B2	$\mathsf{I}_0 \mathbin{+\!\!+} f = f \mathbin{+\!\!+} \mathsf{I}_0$		
B3	$f \circ (g \circ h) = (f \circ g) \circ h$	R2	$(f \uparrow^m) \circ g = (f \circ (g \mathbin{+\!\!+} \mathsf{I}_m)) \uparrow^m$
B4	$\mathsf{I}_m \circ f = f = f \circ \mathsf{I}_n$ for $f : \mathbf{S}^m \to \mathbf{S}^n$	R3	$(f \circ (g \uparrow^m) = (f \mathbin{+\!\!+} g) \uparrow^m$
B5	$(f \mathbin{+\!\!+} f') \circ (g \mathbin{+\!\!+} g') = (f \circ g) \mathbin{+\!\!+} (f' \circ g')$	R4	$(f \circ (\mathsf{I}_l \mathbin{+\!\!+} g)) \uparrow^m = ((\mathsf{I}_k \mathbin{+\!\!+} g) \circ f) \uparrow^n$ for $f : \mathbf{S}^{k+m} \to \mathbf{S}^{l+n}$, $g : \mathbf{S}^n \to \mathbf{S}^m$
B6	$\mathsf{I}_k \mathbin{+\!\!+} \mathsf{I}_l = \mathsf{I}_{k+l}$		
B7	${}^k\!\times^l \circ {}^l\!\times^k = \mathsf{I}_{k+l}$		
B8	${}^k\!\times^0 = \mathsf{I}_k$	R5	$f \uparrow^0 = f$
B9	${}^k\!\times^{l+m} = ({}^k\!\times^l \mathbin{+\!\!+} \mathsf{I}_m) \circ (\mathsf{I}_l \mathbin{+\!\!+} {}^k\!\times^m)$	R6	$(f \uparrow^l) \uparrow^k = f \uparrow^{k+l}$
B10	$(f \mathbin{+\!\!+} g) \circ {}^m\!\times^n = {}^k\!\times^l \circ (g \mathbin{+\!\!+} f)$ for $f : \mathbf{S}^k \to \mathbf{S}^m$, $g : \mathbf{S}^l \to \mathbf{S}^n$	F1	$\mathsf{I}_k \uparrow^k = \mathsf{I}_0$
		F2	${}^k\!\times^k \uparrow^k = \mathsf{I}_k$

Theorem 4.6: Our stream algebra operators together with the operators $(\mathbin{+\!\!+}, \circ, \uparrow, \mathsf{I}, \times)$ forms a data transformer model of BNA and fulfils its main BNA axioms in Table I.

Proof: The proof follows from the ability to describe the main BNA compositional and constant operators using the operators and semantics of our stream algebra. So, our algebra is a BNA model and fulfils its main axioms. We refer the reader to [41] for the proofs of axioms.

B. BNA Synchronous and Asynchronous Operators

The data transformer model of BNA also defines a set of additional higher-order operators for branching, starting, and ending of streams in synchronous and asynchronous data-flow networks. The synchronous operators are defined as follows:

*Definition 4.7 (***High-order Copy):** the operator $\bigwedge^m\!: \mathbf{S}^m \to \mathbf{S}^m \times \mathbf{S}^m$ performs the mapping $\langle x, (x, x)\rangle \in \bigwedge^m$ to duplicate m input streams twice to produce $2m$ output streams. It is expressed in our stream algebra using the special case $\bigwedge^1 = \textsc{copy}(2)$ for $m = 1$, and the general case $\bigwedge^m = \textsc{mult}() \circ \textsc{copy}(2) \circ (\frac{2}{\mathbin{+\!\!+}_{i=1}} \textsc{scatter}_i(\lambda\, z : z, \lambda\, z, j : j))$ for $m \geq 2$. The Mult operator constructs the list $x = \{x^1, .., x^m\}$, where $\forall i \in \{1, .., m\} : x^i \in S^i_{\text{in}}$. Copy then duplicates the input lists into two output streams, which are fed to two Scatter operators to produce $\mathbf{S}^m \times \mathbf{S}^m$ streams. Notice that each $x^i \in S^i_{\text{in}}$ is written to both S^i_{out} and S^{m+i}_{out}.

*Definition 4.8 (***High-order Sink):** the operator $\mathring{\downarrow}^m\!: \mathbf{S}^m \to \varnothing$ ends the incoming streams. It is expressed in our stream algebra using the special case $\mathring{\downarrow}^1 = \textsc{ground}()$, and the general case $\mathring{\downarrow}^m = \frac{m}{\mathbin{+\!\!+}_{i=1}} \textsc{ground}_i()$.

*Definition 4.9 (***High-order Source):** the operator $\mathring{\uparrow}_m\!: \varnothing \to \mathbf{S}^n$ generates n output streams. It is expressed in our stream algebra using the special case $\mathring{\uparrow}_1 = \textsc{source}(u, h)$, and the general case $\mathring{\uparrow}_m = \frac{m}{\mathbin{+\!\!+}_{i=1}} \textsc{source}_i(u_i, h_i)$. The dummy Source operator $\bullet\!\uparrow_m$ is a special version of $\mathring{\uparrow}_m$ that generate m error streams of the constant value `err`.

*Definition 4.10 (***High-order Equality-Test):** the operator $\bigvee_m\!: \mathbf{S}^m \times \mathbf{S}^m \to \mathbf{S}^m$ performs an equality test. The operator has the mapping $\langle (x, z), y\rangle \in \bigvee_m$, where each output $y = (y^1, .., y^m)$ is given by $\forall i \in \{1, ..., m\}$: if $x^i = z^i$ then $y^i = x^i$ else $y^i = \texttt{err}$, and `err` indicates error.

TABLE II: The additional axioms of the synchronous and asynchronous operators of the data transformer model of BNA.

A1	$(\bigvee_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \bigvee_m = (\mathsf{I}_m \mathbin{+\!\!+} \bigvee_m) \circ \bigvee_m$	A5	$\bigwedge^m \circ (\bigwedge^m \mathbin{+\!\!+} \mathsf{I}_m) = \bigwedge^m \circ (\mathsf{I}_m \mathbin{+\!\!+} \bigwedge^m)$
A2	${}^m\!\times^m \circ \bigvee_m = \bigvee_m$	A6	$\bigwedge^m \circ {}^m\!\times^m = \bigwedge^m$
A3	$(\bullet_m \mathbin{+\!\!+} \mathsf{I}_m) \circ \bigvee_m = \mathring{\downarrow}^m \circ \bullet_m$	A7	$\bigwedge^m \circ (\mathring{\downarrow}^m \mathbin{+\!\!+} \mathsf{I}_m) = \mathsf{I}_m$
A4	$\bigvee_m \circ \mathring{\downarrow}^m = \mathring{\downarrow}^m \mathbin{+\!\!+} \mathring{\downarrow}^m$	A8	$\bullet_m \circ \bigwedge^m = \bullet_m \mathbin{+\!\!+} \bullet_m$
		A9	$\mathring{\uparrow}_m \circ \mathring{\downarrow}^m = \mathsf{I}_0$
		A10	$\bigvee_m \circ \bigwedge^m = (\bigwedge^m \mathbin{+\!\!+} \bigwedge^m) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\!\times^m \mathbin{+\!\!+} \mathsf{I}_m) \circ (\bigvee_m \mathbin{+\!\!+} \bigvee_m)$
		A11	$\bigwedge^m \circ \bigvee_m = \mathsf{I}_m$
A12	$\mathring{\uparrow}_0 = \mathsf{I}_0$	A16	$\mathring{\downarrow}^0 = \mathsf{I}_0$
A13	$\mathring{\uparrow}_{m+n} = \mathring{\uparrow}_m \mathbin{+\!\!+} \mathring{\uparrow}_n$	A17	$\mathring{\downarrow}^{m+n} = \mathring{\downarrow}^m \mathbin{+\!\!+} \mathring{\downarrow}^n$
A14	$\bigvee_0 = \mathsf{I}_0$	A18	$\bigwedge^0 = \mathsf{I}_0$
A15	$\bigvee_{m+n} = (\mathsf{I}_m \mathbin{+\!\!+} {}^n\!\times^m \mathbin{+\!\!+} \mathsf{I}_n) \circ (\bigvee_m \mathbin{+\!\!+} \bigvee_n)$	A19	$\bigwedge^{m+n} = (\bigwedge^m \mathbin{+\!\!+} \bigwedge^n) \circ (\mathsf{I}_m \mathbin{+\!\!+} {}^m\!\times^n \mathbin{+\!\!+} \mathsf{I}_n)$
F3	$\bigvee_m \uparrow^m = \mathring{\downarrow}^m$	F4	$\bigwedge^m \uparrow^m = \bullet_m$

The asynchronous versions of the additional higher-order constant operators are defined by [41], [42] similar to the definition of the synchronous constants; however, they produce asynchronous output streams. We can describe them in our stream algebra by applying a set of parallel CUT or Latch. For example, $\textsc{parallel-cut}(m) = \frac{m}{\mathbin{+\!\!+}_{i=1}} \textsc{cut}^G_i()$ receive m synchronous input streams and produce m asynchronous output streams. This allows us to define the asynchronous higher-order Cut-Copy as $\overline{\bigwedge}^m = \bigwedge^m \circ \textsc{parallel-cut}(2m)$. The higher order Cut-Equality-Test and Cut-Source operators are defined similarly as $\overline{\bigvee}_m = \bigvee^m \circ \textsc{parallel-cut}(m)$, and $\overline{\mathring{\uparrow}}_m = \mathring{\uparrow}_m \circ \textsc{parallel-cut}(m)$, respectively. We can also define the Latch version of the operators $(\widehat{\bigwedge}_m, \widehat{\bigvee}_m, \widehat{\mathring{\uparrow}}_m)$ by replacing PARALLEL-CUT with PARALLEL-LATCH. Notice that Sink has the same definition whether the incoming streams are synchronous or asynchronous.

Theorem 4.11: Our stream algebra implements the extended synchronous and asynchronous operators of the data transformer model of BNA and fulfils its additional BNA axioms in Table II.

Proof: The proof again follows from the ability to describe the extended synchronous and asynchronous operators of the the data transformer model of BNA using the operators and semantics of our stream algebra. It was shown by [41], [42] that the axioms in Table II are valid for both the synchronous and asynchronous operators. We refer the reader to [41] for the proofs of the additional axioms.

C. Axioms of the Stream Algebra

Theorems 4.6 and 4.11 provide several advantages to our stream algebra. They show that our stream algebra fulfills the BNA axioms of [41], [42]. These axioms define equivalence relations between different combinations of stream operators, and provide a way to determine equivalence between data-flow graphs, and to simplify complex ones.

Although our stream algebra fulfils the data transformer model of BNA, the algebra provides a larger set of branching and data transforming operators. In this section, we present a set of valid axioms for our stream algebra operators.

TABLE III: The axioms of our stream algebra.

S1	$\text{MAP}(f) \circ \text{MAP}(h) = \text{MAP}(f \cdot h), \quad \text{for } f \cdot h(x) = f(h(x))$
S2	$\text{MAP}(f) \circ \text{REDUCE}(g) = \text{REDUCE}(g'), \text{ for } g'(x,y) = g(f(x),u)$
S3	$\text{REDUCE}(g) \circ \text{MAP}(f) = \text{REDUCE}(g'), \text{ for } g'(x,u) = \{y, u = g(x,u) \text{ return}(f(y),u)\}$
S4	$\text{MULT}() \circ \text{SCATTER}(\lambda x : x, \lambda x_i, i : i) = \mathsf{I}_m, \text{ for MULT}() : \mathbf{S}^m \to \mathbf{S}$
S5	$\text{FILTER}(\theta) \circ \text{ADD}() = \mathsf{I}_1$
S6	${}^m\!\times^n \circ \text{ADD}() = {}^n\!\times^m \circ \text{ADD}()$
S7	${}^m\!\times^n \circ \text{MERGE}() = {}^n\!\times^m \circ \text{MERGE}()$
S8	$\text{COPY}(m+n) \circ {}^m\!\times^n = \text{COPY}(m+n)$
S9	$\text{SOURCE}() \circ \text{GROUND}() = \varnothing$
S10	$h() \circ \|_{i=1}^{m} \text{GROUND}_i() = \|_{i=1}^{n} \text{GROUND}_i(), \text{ for } h : \mathbf{S}^n \to \mathbf{S}^m \text{ and } h \notin \{\text{MAP}, \text{REDUCE}\}$
S11	$\text{COPY}(m+n) \circ (\mathsf{I}_m \mathbin{+\!\!+} \|_{i=1}^{n} \text{GROUND}_i()) = h$ where $h = \text{COPY}(m)$ if $m > 1$, and $h = \mathsf{I}_1$ if $m = 1$
S12	$\text{LEFT-MULT}(k) = {}^m\!\times^k \circ \text{RIGHT-MULT}(k) \circ \text{MAP}(f)$ for $\text{LEFT-MULT}(k) : \mathbf{S}^{m+k} \to \mathbf{S}, \text{RIGHT-MULT}(k) : \mathbf{S}^{k+m} \to \mathbf{S}$ for $f = \lambda x : (x^{k+1}, ...x^{k+m}, x^1, ..., x^k), \|x\| = k+m$
S13	$(\text{MERGE}_1(f) \mathbin{+\!\!+} \mathsf{I}_n) \circ \text{MERGE}_2(f) = (\mathsf{I}_m \mathbin{+\!\!+} \text{MERGE}_3(f)) \circ \text{MERGE}_4(f)$ for $\text{MERGE}_1(f) : \mathbf{S}^m \to \mathbf{S}$, $\text{MERGE}_2(f) : \mathbf{S}^{n+1} \to \mathbf{S}$ for $\text{MERGE}_3(f) : \mathbf{S}^n \to \mathbf{S}$, $\text{MERGE}_4(f) : \mathbf{S}^{m+1} \to \mathbf{S}$
S14	$(\text{MERGE}_1(f) \mathbin{+\!\!+} \text{MERGE}_{2(f)}) \circ \text{MERGE}_3(f) = \text{MERGE}_4(f)$ for $\text{MERGE}_1(f) : \mathbf{S}^m \to \mathbf{S}$, $\text{MERGE}_2(f) : \mathbf{S}^n \to \mathbf{S}$ for $\text{MERGE}_3(f) : \mathbf{S}^2 \to \mathbf{S}$, $\text{MERGE}_4(f) : \mathbf{S}^m \times \mathbf{S}^n \to \mathbf{S}$
S15	$\text{COPY}(n) \circ (\text{COPY}(m) \mathbin{+\!\!+} \mathsf{I}_{n-1}) = \text{COPY}(n) \circ (\mathsf{I}_{n-1} \mathbin{+\!\!+} \text{COPY}(m))$

Theorem 4.12: Our algebra fulfils the axioms in Table III.

Proof: The axioms S1, S2, and S3 clearly hold. They define composition relationships between data transforming operators, which enable merging these operators to eliminate large communication latencies in data streaming pipelines. The axioms S4 to S15 are inferred from the additional BNA axioms of Table II. The axioms S4 and S5 are similar to A11. For S4, the left side performs the mapping $\langle x, x\rangle \in \mathbf{S}^m \to \mathbf{S}^m$, which is I_m. For S5, $\text{FILTER}(\theta) : S_{\text{in}} \to S^1, S^2$ produces the sequence of mappings $\{\langle x_i, x_i\rangle \in S_{\text{in}} \to S^j, \langle x_{i+1}, x_{i+1}\rangle \in S_{\text{in}} \to S^l\}$ for any $\{x_i, x_{i+1}\} \subset S_{\text{in}}$, and $j, l \in \{1, 2\}$. This implies that $\text{FILTER}(\theta) \circ \text{ADD}() : S_{\text{in}} \to S_{\text{out}}$ produces $\{x_i, x_{i+1}\} \subset S_{\text{out}}$ and $S_{\text{in}} = S_{\text{out}}$. The axioms S6 and S7 are similar to A2, and they show the commutativity of the Add and Merge operators towards their input streams. S6 holds by Definition 3.12, where Add will produce same output for any permutation of the incoming streams. Similarly, S7 holds by Definition 3.14 as Merge will maintain its partial order for any permutation of the incoming streams. S8 and S9 clearly hold, and they are deduced from A6 and A9, respectively. S10 is a generalization of A4, and is obvious and valid for all operators except Map and Reduce, which perform data transformations. S11 is similar to A7, where grounding n output streams of $\text{COPY}(m+n)$ is equivalent to $\text{COPY}(m)$ when $m > 1$, and is equivalent to I_1 when $m = 1$. S12 is inspired by A2 and A6, and allows writing Left-Mult using Right-Mult and vice versa. The leftside performs the mapping $\langle (x, z), y\rangle \in \text{LEFT-MULT}(k)$, where $y = \{x^1, ..., x^m, w^1, .., w^k\}$. We have $\langle (x, z), y'\rangle \in {}^m\!\times^k \circ \text{RIGHT-MULT}(k)$, where $y' = (w^1, ..., w^k, x^1, .., x^m)$, and $\langle y', y\rangle \in \text{MAP}(f)$. S13 is inferred from A1 and shows the associativity of the Merge operator. The axiom has the left and right expressions performing the mapping $\langle (x, z), y^*\rangle \in \mathbf{S}^m \times \mathbf{S}^n \to \mathbf{S}$. The left expression performs the minimum ordering $y^* = \min(\min_{1\le j\le m}(f(x^j)), f(z^1), ..., f(z^n))$, which is equivalent to the right expression that has $y^* = \min(f(x^1), ..., f(x^m), \min_{1\le i\le n}(f(z^i)))$. Notice that the associativity of the Add operator is not guaranteed because of its non-deterministic behaviour. S14 is deduced from A15 and the associativity and commutativity of the Merge operator. This axiom has both sides performing the mapping $\langle (x, z), y^*\rangle \in \mathbf{S}^m \times \mathbf{S}^n \to \mathbf{S}$. The left side performs the minimum ordering $y^* = \min(\min_{1\le j\le m}(f(x^j)), \min_{1\le i\le n}(f(z^i)))$, which is equivalent to the right side that has $y^* = \min(f(x^1), ..., f(x^m), f(z^1), ..., f(z^n))$. S15 is similar to A5, and is clearly valid with both the left and right sides performing the mapping $\left\langle x, (x^{\overset{\times m+n-1}{\cdots\cdots}})\right\rangle \in \mathbf{S} \to \mathbf{S}^{m+n-1}$.

V. Examples

There is considerable interest in developing online computer vision algorithms that use feedback control to perform parameter tuning or iterative optimization tasks. These tasks allow an online algorithm to adapt itself continuously to different scene contexts or iteratively improve output results over time. In this section, we discuss two state-of-the-art algorithms [36], [10] that process vision streams and apply feedback-control to perform parameter tuning and iterative optimization. Without loss of generality, we will discuss how we can effectively express the feedback-control of these algorithms using our stream algebra.

A. Iterative Optimization for Aligning Photo Streams

Nowadays, several photo hosting websites store vast amounts of personal image and video collections. These visual collections are usually received and stored as photo streams organized in chronological order. Kim et al. [10] developed a method that processes Flickr photo streams to build common storylines. The method works by iterating between the following tasks: an alignment task and a co-segmentation task. To describe the iterative optimization algorithm by [10] using our stream algebra, the following data types are defined:

$$\texttt{Shape} : \text{LIST}\left\langle \mathbb{R}^2 \right\rangle; \quad \texttt{Region} : \texttt{Shape} \times \texttt{Histogram}$$
$$\texttt{Photo} : \texttt{2DImage} \times \texttt{Time} \times \texttt{Histogram} \times \text{LIST}\left\langle \texttt{Region} \right\rangle \times \mathbb{R}$$

$\texttt{Block} : \textsc{List}\langle\texttt{Photo}\rangle \times \texttt{Time}^2$;
$\texttt{BlocksList} : \textsc{List}\langle\texttt{Block}\rangle \times \mathbb{R}$;
$\texttt{BlocksGraph} : \textsc{Graph}\langle\texttt{Block}, \texttt{Block} \times \texttt{Block}\rangle$;
$\texttt{PhotosGraph} : \textsc{Graph}\langle\texttt{Photo}, \texttt{Photo} \times \texttt{Photo}\rangle$;
$\texttt{BlocksStruct} : \texttt{BlocksGraph} \times \mathbb{R}$;
$\texttt{PhotosStruct} : \texttt{BlocksStruct} \times \texttt{PhotosGraph}$

A type Shape defines a 2D point list. A Region is a vector (s, b), where s : Shape is the region boundary and b : Histogram is the feature descriptor of the region. A Photo is a vector (a, t, b, r, nn), where a : 2DImage is a 2D image, t : Time is the capture time of the photo, b : Histogram is a feature descriptor, $r : \textsc{List}\langle\texttt{Region}\rangle$ is a list of computed foreground regions, and nn is a pointer to the nearest-neighbour photo. A Block is a vector (b, t_1, t_2), where $b : \textsc{List}\langle\texttt{Photo}\rangle$. In addition, t_1 : Time and t_2 : Time are the earliest and latest capture times for all photos in b. A BlocksList is a vector (w, itr), where $w : \textsc{List}\langle\texttt{Block}\rangle$, and $itr : \mathbb{R}$ is a variable recording the iteration number. A BlocksStruct is a vector (c_1, itr), where c_1 is a graph with blocks as vertices. A PhotosStruct is a vector (q, c_2), where q : BlocksStruct, and c_2 is a graph with photos as vertices. We assume n input photo streams $I = \{I_i | i \leq n\}$. To start processing the input streams, we define the function $g_1 : \texttt{Block} \times \texttt{Photo} \to \texttt{Block} \times \texttt{Block}$:

$$
\begin{aligned}
g_1(u, x) = \quad \{ \quad & \text{if duration}(u) \geq \triangle t_1 \text{ then} \\
& \quad u' = \varnothing; y = u \\
& \text{else} \\
& \quad u' = u \oplus x \quad \text{//append } x \text{ to Block } u \\
& \quad y = \varnothing \\
& \text{return}(u', y) \quad \}
\end{aligned}
$$

The g_1 function partitions an incoming stream of photos into a set of blocks. The function buffers the incoming photos into a state variable u that defines a block. While buffering, the output y is set to an empty block. The buffering continues until the block u has its duration larger than a predefined interval $\triangle t_1$ (24 hours as defined by [10]). We can then process the input photo streams using the composite operator $h_1 : \mathbf{S}^n\langle\texttt{Photo}\rangle \to \mathbf{S}^n\langle\texttt{Block}\rangle$:

$$
\begin{aligned}
h_1 = {}_{i=1}^{n}\Big(&\textsc{reduce}(g_1, \text{Empty-List}) \\
&\circ \textsc{filter}(\lambda\, x :\ |x| \neq 0) \circ (\mathsf{l}_1 +\!\!+ \textsc{ground}())\Big)_i .
\end{aligned} \tag{3}
$$

The operator h_1 starts by applying a Reduce operator parameterized by the function g_1 on each input stream I_i to produce a stream of photo blocks. A Filter operator followed by $(\mathsf{l}_1 +\!\!+ \textsc{ground}())$ remove empty blocks and produce the blocks stream $B_i : \mathbf{S}\langle\texttt{Block}\rangle$, where $B_1, ..., B_n \triangleq h_1()(I_1, ..., I_n)$. The block streams are then forwarded to the composite operator $h_2 : \mathbf{S}^n\langle\texttt{Block}\rangle \to \mathbf{S}\langle\texttt{BlocksList}\rangle$:

$$
h_2 = \textsc{mult}() \circ \textsc{map}(f_7). \tag{4}
$$

The operator h_2 applies a Mult operator to synchronize the block streams $\{B_i | i \leq n\}$, and produce the block-list stream $L : \mathbf{S}\langle\textsc{List}\langle\texttt{Block}\rangle\rangle$. A Map operator then applies the function $f_7 : \textsc{List}\langle\texttt{Block}\rangle \to \texttt{BlocksList}$ on every list $l_i \in L$ to select blocks that overlap in time by a period of at least $\triangle t_2$ (an hour as defined by [10]). This produces the stream $Q : \mathbf{S}\langle\texttt{BlocksList}\rangle$, where $Q \triangleq h_2()(B_1, ..., B_n)$. The Q stream is then forwarded to the composite operator $h_3 : \mathbf{S}\langle\texttt{BlocksList}\rangle \times \mathbf{S}\langle\texttt{BlocksList}\rangle \to \mathbf{S}\langle\texttt{PhotosStruct}\rangle$:

$$
\begin{aligned}
h_3 = \big(&\textsc{add}() \circ \textsc{map}(f_8) \circ \textsc{map}(f_9) \circ \textsc{filter}(\theta) \\
&\circ (\mathsf{l}_1 +\!\!+ \textsc{map}(f_{10}))\big) \uparrow^1 .
\end{aligned} \tag{5}
$$

The operator h_3 receives the feedback stream $F : \mathbf{S}\langle\texttt{BlocksList}\rangle$ and the stream Q. These streams are added together into the stream $P : \mathbf{S}\langle\textsc{List}\langle\texttt{BlocksList}\rangle\rangle$, where $P \triangleq \textsc{add}()(Q, F)$. The stream P is then processed using two map operators that apply the functions $f_8 : \texttt{BlocksList} \to \texttt{BlocksStruct}$ and $f_9 : \texttt{BlocksStruct} \to \texttt{PhotosStruct}$. The function f_8 converts the list of blocks in each element $x_j \in P$ into a graph $y_j : \text{Graph}\langle\texttt{Block}, \texttt{Block} \times \texttt{Block}\rangle$. An edge exists between two blocks in y_j if the two blocks have the smallest distance between each other. The function f_9 converts a graph of blocks to a graph of photos and increments the iteration counter variable by 1. It also applies co-segmentation on the graph of photos to assign each photo a set of foreground regions or improve an existing one. The stream $M : \mathbf{S}\langle\texttt{PhotosStruct}\rangle$ is then defined by $M \triangleq \textsc{map}(f_8) \circ \textsc{map}(f_9)(P)$, and forwarded to a Filter operator that applies the predicate $\theta = \lambda\, x :\ x.q.itr \geq N_{stop}$ on every $x \in M$. This generates the streams $M', R \triangleq \textsc{filter}(\theta)(M)$. A function $f_{10} : \texttt{PhotosStruct} \to \texttt{BlocksList}$ is then defined to convert the data type of elements in R to BlocksList. This is by copying the iteration number and generating a list of photo blocks from the vertices of the blocks graph of every $x \in R$. The output and feedback streams are obtained by $M', F \triangleq h()(M', R)$, where $h = (\mathsf{l}_1 +\!\!+ \textsc{map}(f_{10}))$, and $F : \mathbf{S}\langle\texttt{BlocksList}\rangle$. The stream F is fed back by $\uparrow^1$, and merged with the stream Q using the Add operator to produce the stream P.

The entire pipeline of [10] can be then described using the composite operator $h : \mathbf{S}^n\langle\texttt{Photo}\rangle \to \mathbf{S}\langle\texttt{PhotosStruct}\rangle$:

$$
h = h_1 \circ h_2 \circ h_3. \tag{6}
$$

B. Road Boundary Detection

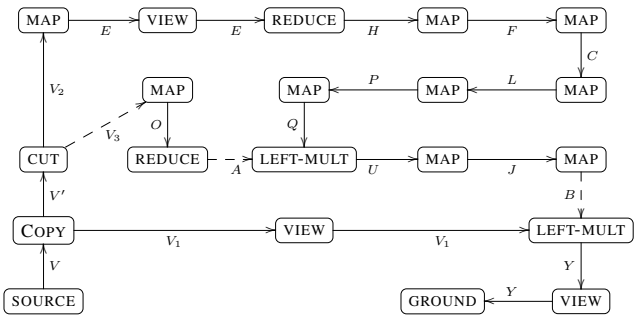

Fig. 3: Online road-boundary detection algorithm [28] described in the stream algebra. The workflow graph with arrows showing the flow direction of streams. Letters on arrows represent stream names. Dashed lines indicate decoupled streams. The input video stream is V, and the output video stream is Y.

In this section, we express our online road-boundary detection algorithm [28] in the proposed stream algebra. The algorithm receives an input video stream $V = v_i | i \geq 0\}$. Then,

it applies edge detection on each frame $v_i \in V$ by extracting N superpixels from each v_i and applying polygon approximation. The resulting edges are incrementally added to a hierarchical-clustering tree by applying an online algorithm that maintains clusters over a temporal window of interval Δt. The algorithm generates a sequence of updated clustering trees $T = \{t_i | i \geq 0\}$. For each tree t_i, the algorithm statistically ranks the clusters based on the number of edges and variance. Clusters with ranks larger than a threshold τ are then selected. This generates a ranked cluster stream $C = \{c_i | i \geq 0\}$, where each $c_i \in C$ is a list of top-ranked clusters from $t_i \in T$ at time i. For every list c_i, each cluster $x \in c_i$ is mapped to its mean line. This generates the line stream $L = \{l_i | i \geq 0\}$. The method then performs a Cartesian product of each set $l_i \in L$ by itself and eliminates pairs with similar elements. This generates a pairwise stream $P = \{p_i | i \geq 0\}\}$. After that, the approach applies perspective filtering on every $p_i \in P$ to remove line pairs that do not have their vanishing points heading upward in the image. This generates the filtered stream of line pairs Q.

After generating the stream Q, the algorithm ranks every pair $q_i \in Q$ based on the road activity. This is performed by taking the input stream V and applying background subtraction to detect moving objects. The centroids of these objects are recorded over a temporal window of the same interval Δt used by online hierarchical clustering. Then, the method attaches the recent list of detected centroids with every line-pair list q_i to produce the stream $U = \{u_i | i \geq 0\}$. Next, activity ranking is applied on U to construct the ranked pairwise stream J. Finally, the algorithm outputs the dominant road-boundary stream $B = \{b_i | i \geq 0\}$, where $b_i = \arg\max_{x \in y_i} rank(x)$ represents the top-ranked pair from every pairwise list $y_i \in J$.

Now, we describe this vision pipeline using the stream algebra (Figure 3). The data types are defined as:

$$\begin{aligned}
&\texttt{Frame}: \texttt{2DImage}; \quad \texttt{Point}: \mathbb{R}^2 \\
&\texttt{Edge}: \mathbb{R}^6; \quad \texttt{Cluster}: \mathbb{R}^4 \times \texttt{Edge} \\
&\texttt{RCluster}: \texttt{Cluster} \times \mathbb{R} \\
&\texttt{Pair}: \texttt{Edge} \times \texttt{Edge}; \quad \texttt{RPair}: \texttt{Pair} \times \mathbb{R} \\
&\texttt{Hierarchy}: \textsc{Tree}\langle \texttt{Cluster} \rangle \\
&\texttt{Params}: \textsc{List}\langle \texttt{Parameter} \rangle,
\end{aligned}$$

where a `Frame` is a single 2D image, a `Point` is a 2D vector, and an `Edge` is a straight line segment $(x_1, x_2, y_1, y_2, \rho, \phi)$, where (ρ, ϕ) represents the edge in polar coordinates. A `Cluster` is represented in sufficient statistics $(\widehat{\phi}, \widehat{\rho}, n, t, s_{max})$, where:

$$\widehat{\phi} = (\sum_{i=0}^{n} \phi_i, \sum_{i=0}^{n} \phi_i^2), \quad \widehat{\rho} = (\sum_{i=0}^{n} \rho_i, \sum_{i=0}^{n} \rho_i^2),$$

n is the number of edges in the cluster, t is the last update time of the cluster, and s_{max} is the line segment that encloses the projection of all cluster edges on its mean line. We define `RCluster` as (c, α), where c : `Cluster` and α is the statistical rank of c. A `Pair` is a pair of edge segments. The `RPair` is defined as (p, β), where p : `Pair`, and β is the activity rank of p. A `Hierarchy` is a tree of clusters. Finally, `Params` is a list of parameters.

We start by generating a video stream $V : \mathbf{S}\langle \texttt{Frame} \rangle$ using a Source operator parameterized by the generator function f, where $V \triangleq \textsc{source}(\text{Empty-List}, f)$. This stream is forwarded to the composite operator $h_1 : \mathbf{S}\langle \texttt{Frame} \rangle \rightarrow \mathbf{S}^3\langle \texttt{Frame} \rangle$:

$$h_1 = \textsc{copy}(2) \circ (\textsc{map}(d_1) +\!\!+ \textsc{cut}()), \tag{7}$$

which applies a Copy operator to duplicate V to the streams V_1 and V', and process them in parallel. The stream V_1 is forwarded to a Map operator that acts as a viewer to display the video stream using the function $d_1 : \texttt{Frame} \rightarrow \texttt{Frame}$. The function d_1 forwards its input to output while displaying the image content of the incoming elements. We then apply the CUT operator on V' to obtain the streams V_2 and V_3, where $V_1, V_2, V_3 \triangleq h_1(V)$. Note that the streams V_1, V_2, and V_3 are all copies of the input stream V; however, V_3 is an asynchronous copy, whereas, V_1 and V_2 are synchronous copies.

Next we define the composite operator $h_2 : \mathbf{S}\langle \texttt{Frame} \rangle \rightarrow \mathbf{S}\langle \textsc{List}\langle \texttt{RCluster} \rangle \rangle$:

$$\begin{aligned}
h_2 = {} & \textsc{map}(f_1, q_1) \circ \textsc{map}(d_2) \\
& \circ \textsc{reduce}(\text{Empty-Tree}, g_1, z_1) \circ \textsc{map}(f_2),
\end{aligned} \tag{8}$$

which process the stream V_2 by applying a sequence of Map and Reduce operators. The first Map operator is parameterized by the function $f_1 : \texttt{Frame} \times \texttt{Params} \rightarrow \textsc{List}\langle \texttt{Edge} \rangle$, which performs superpixel segmentation and contour approximation on each frame $x \in V_2$. The function f_1 takes a list of parameters q_1 that define the number of superpixels N. This generates the stream $E : \mathbf{S}\langle \textsc{List}\langle \texttt{Edge} \rangle \rangle$, where $E \triangleq \textsc{map}(f_1, q_1)(V_2)$. This stream is displayed by the second Map operator using the function d_2. The stream E is then forwarded to a Reduce operator parameterized by the function $g_1 : \texttt{Hierarchy} \times \textsc{List}\langle \texttt{Edge} \rangle \times \texttt{Params} \rightarrow \texttt{Hierarchy} \times \textsc{List}\langle \texttt{Cluster} \rangle$, an initial empty clustering tree, and a set of parameters z_1. The function g_1 is defined as:

$$\begin{aligned}
g_1(u, x, p) = \{ \; & u.\text{add}(x); && \text{//add edges } x \text{ to } u \\
& y = u.\text{updated}(); && \text{//get updated clusters.} \\
& \text{return}(u, y) \quad \}, &&
\end{aligned}$$

which keeps updating a given clustering tree u by adding new edges. Then, a list of all clusters updated by the added edges are returned as the output y. The list z_1 contains several parameters such as tree height and number of children per node. The Reduce operator generates the stream $H : \mathbf{S}\langle \textsc{List}\langle \texttt{Cluster} \rangle \rangle$. This stream is fed to another Map operator parameterized by a ranking function $f_2 : \textsc{List}\langle \texttt{Cluster} \rangle \times \texttt{Params} \rightarrow \textsc{List}\langle \texttt{RCluster} \rangle$, which statistically rank clusters based on the variance and number of samples. The stream $F : S\langle \textsc{List}\langle \texttt{RCluster} \rangle \rangle$ is then defined as $F \triangleq \textsc{Map}(f_2)(H)$, and also $F \triangleq h_2()(V_2)$.

The stream F is then forwarded to the composite operator $h_3 : \mathbf{S}\langle \textsc{List}\langle \texttt{RCluster} \rangle \rangle \rightarrow \mathbf{S}\langle \textsc{List}\langle \texttt{Pair} \rangle \rangle$:

$$h_3 = \textsc{map}(f_3, q_2) \circ \textsc{map}(f_4) \circ \textsc{map}(f_5) \circ \textsc{map}(f_6), \tag{9}$$

which applies a sequence of Map operators. The first Map operator is parameterized by the threshold function $f_3 : \textsc{List}\langle \texttt{RCluster} \rangle \times \texttt{Params} \rightarrow \textsc{List}\langle \texttt{RCluster} \rangle$, and the set of parameters q_2. The function f_3 selects clusters with ranks larger than a threshold τ, which is defined by the list q_2. The stream $C : \mathbf{S}\langle \textsc{List}\langle \texttt{RCluster} \rangle \rangle$ is then defined as $C \triangleq \textsc{map}(f_3, q_2)(F)$. This stream is converted to a line stream by applying a function $f_4 : \textsc{List}\langle \texttt{RCluster} \rangle \times \texttt{Params} \rightarrow \textsc{List}\langle \texttt{Edge} \rangle$, which maps each cluster $x \in c$ into its mean line, and $c \in C$. The line stream $L : \mathbf{S}\langle \textsc{List}\langle \texttt{Edge} \rangle \rangle$ is then

constructed as $L \triangleq \text{MAP}(f_4)(C)$. For every $l \in L$, a Cartesian product function $f_5 : \text{LIST}\langle \texttt{Edge} \rangle \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{Pair} \rangle$ obtains the product of l by itself, and removes pairs with similar elements. The pairwise stream $P : S\langle \text{LIST}\langle \texttt{Pair} \rangle \rangle$ is then defined as $P \triangleq \text{MAP}(f_5)(L)$. Next, we define the filtering function $f_6 : \text{LIST}\langle \texttt{Pair} \rangle \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{Pair} \rangle$ that applies perspective filtering on every pair $y \in p$, and $p \in P$. This function returns a list that only contains pairs with vanishing points heading upward in the image. We then define the stream $Q : S\langle \text{LIST}\langle \texttt{Pair} \rangle \rangle$, where $Q \triangleq \text{MAP}(f_6)(P)$, and also $Q \triangleq h_3()(F)$. We can also define $Q \triangleq h_4(V_2)$, where the composite operator $h_4 : \mathbf{S}\langle \texttt{Frame} \rangle \rightarrow \mathbf{S}\langle \text{LIST}\langle \texttt{Pair} \rangle \rangle$ is defined as:

$$h_4 = h_2 \circ h_3. \tag{10}$$

To describe scene activity, we define the composite operator $h_5 : \mathbf{S}\langle \texttt{Frame} \rangle \rightarrow \langle \text{LIST}\langle \texttt{Point} \rangle \rangle$:

$$h_5 = \text{MAP}(f_7, q_3) \circ \text{REDUCE}(\text{Empty-List}, g_2, z_2), \tag{11}$$

which receives the video stream V_3 as input, and applies a Map followed by Reduce. Remember that the V_3 stream is the asynchronous copy of the input video stream. The Map operator is parameterized by the function $f_7 : \texttt{Frame} \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{Point} \rangle$, which applies background subtraction [43] on every frame $x \in V_3$ to obtain a set of foreground regions, and output their centroids. This generates the centroid stream $O : \mathbf{S}\langle \text{LIST}\langle \texttt{Point} \rangle \rangle$. The Reduce operator is parameterized by the function $g_2 : \text{LIST}\langle \texttt{Point} \rangle \times \text{LIST}\langle \texttt{Point} \rangle \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{Point} \rangle \times \text{LIST}\langle \texttt{Point} \rangle$, and the list of parameters z_2. The function g_2 records centroids over a temporal window of interval $\triangle t$, which is defined by the list z_2. The function g_2 is defined as:

$$\begin{aligned} g_2(u, x, p) = \ \{ \ & \text{for all } z \in u \\ & \quad \text{if } (\text{now}() - .(z) \geq p.\triangle t) \text{ then} \\ & \quad\quad u = u \ominus z \quad \text{//remove } z \text{ from } u \\ & \quad u = u \oplus x.v \quad \text{//append points } x.v \text{ to } u \\ & \quad \text{return}(u, u) \ \}. \end{aligned}$$

The Reduce operator generates the activity stream $A : S\langle \text{LIST}\langle \texttt{Point} \rangle \rangle$, where $A \triangleq h_5()(V_3)$. This stream together with the filtered pairwise stream Q are forwarded to the composite operator $h_6 : \mathbf{S}\langle \text{LIST}\langle \texttt{Pair} \rangle \rangle \times \mathbf{S}\langle \text{LIST}\langle \texttt{Point} \rangle \rangle \rightarrow \mathbf{S}\langle \text{LIST}\langle \texttt{RPair} \rangle \rangle$, which is defined as:

$$h_6 = \text{LEFT-MULT}() \circ \text{MAP}(f_8) \circ \text{MAP}(f_9). \tag{12}$$

The operator h_6 applies a Left-Mult operator to merge the streams Q and A into the stream $U : S\langle \text{LIST}\langle \texttt{Pair} \rangle \times \text{LIST}\langle \texttt{Point} \rangle \rangle$, where $U \triangleq \text{LEFT-MULT}()(Q, A)$. A Map operator then applies a ranking function $f_8 : \text{LIST}\langle \texttt{Pair} \rangle \times \text{LIST}\langle \texttt{Point} \rangle \times \texttt{Params} \rightarrow \text{LIST}\langle \texttt{RPair} \rangle$ to rank every line pair using its attached centroids, and generate a list of ranked pairs. This builds the ranked pairwise stream $J : S\langle \text{LIST}\langle \texttt{RPair} \rangle \rangle$. The algorithm then applies another Map operator using the function $f_9 = \lambda x : \arg\max_{y \in x} r_{\text{Activity}}(y)$ to return the line pair with the maximum activity rank. This construct the dominant road-boundary stream $B : \mathbf{S}\langle \texttt{RPair} \rangle$, where $B \triangleq \text{MAP}(f_9)(J)$. The stream B together with the stream V_1 are forwarded to the composite operator $h_7 : \mathbf{S}\langle \texttt{Frame} \rangle \times \mathbf{S}\langle \texttt{RPair} \rangle \rightarrow \mathbf{S}\langle \texttt{Frame} \times \texttt{RPair} \rangle$, which is defined as:

$$h_7 = \text{LEFT-MULT} \circ \text{MAP}(d_3). \tag{13}$$

This operator merges the streams V_1 and B using a Left-Mult operator into the stream $Y : \mathbf{S}\langle \texttt{Frame} \times \texttt{RPair} \rangle$, where $Y \triangleq \text{LEFT-MULT}()(V_1, B)$. Next a Map operator view the estimated dominant road boundary on every frame using the display function d_3.

We can define the complete pipeline of [28] as the composite operator $h_8 : \mathbf{S}\langle \texttt{Frame} \rangle \rightarrow \mathbf{S}\langle \texttt{Frame} \times \texttt{RPair} \rangle$. Also, we can define $h' : \varnothing \rightarrow \varnothing$ to apply a final Ground operator that releases the stream resources:

$$h_8 = h_1 \circ (\mathrm{I}_1 +\!\!+ ((h_4 +\!\!+ h_5) \circ h_6)) \circ h_7 \tag{14}$$

$$h' = \text{SOURCE}(\text{Empty-List}, f) \circ h_8 \circ \text{GROUND}(). \tag{15}$$

VI. Parmeter Tuning of Large-Scale Computer Vision Systems

In this section, we present a formal method for adaptive parameter tuning of large-scale computer vision pipelines. Specifically, we show that a general optimizer of numerical parameters, such as the method by [29], can be extended using the feedback-control mechanisms of our stream algebra to provide common online parameter optimization for computer vision pipelines. Without loss of generality, the streaming pipeline described in Section V-B for automatic road-boundary detection algorithm will be used as a case study.

A. Problem Formulation

Given the linear computer vision pipeline $h = \bigcirc_{i=1}^{k} h_i$ in Figure 2a, we assume that each data-processing operator h_i executes a user-defined function with a set of input parameters P_i. The parameter settings of the pipeline are defined as $\theta = \bigcup_{i \in [1,n]} P_i$, where $\theta \in \Theta$, and Θ is the parameter configuration space for the pipeline h. The processing functions are defined arbitrarily with no closed-form representation. This requires the computation of gradients to optimize the parameters. The functions may also be expensive to compute. Given these assumptions, we follow the general definition of the algorithm configuration problem [44], [45] to define the pipeline configuration problem as follows:

Definition 6.1 (Pipeline configuration problem): Let a pipeline h have a distribution $\mathcal{I}$ of input instances and a target performance metric $c(\pi, \theta)$ with $\theta \in \Theta$ and on instances $\pi \subset \mathcal{I}$. Let $f(\theta) = E_{\pi \subset \mathcal{I}}[c(\pi, \theta)]$ define the expected performance of the pipeline h using the parameter setting θ on instances π drawn from $\mathcal{I}$. The pipeline configuration problem aims to find a good parameter configuration $\hat{\theta} = \text{argmax}_{\theta \in \Theta}\{f(\theta)\}$, that approximates the best configuration.

Sequential model-based optimization (SMBO) is a popular approach for solving general algorithm configuration problems by optimizing expensive blackbox functions [29], [46], [47], [45]. This approach optimizes the target performance function $f : \Theta \rightarrow \mathbb{R}$ by sequentially evaluating samples of the parameter space Θ, while minimizing the number of samples required to reach a good parameter setting. This is performed by first calculating $y = f(\theta)$ on different values of θ, and on the set of input data instances. This generates the initial set $\mathcal{D} = \{(y_i, \theta_i) | 1 \leq i \leq n\}$, which is referred to as the initial design.

The set $\mathcal{D}$ is used to learn a Gaussian process (GP) model that defines a probability distribution f over a continuous

range of $\theta \in \Theta$. The SMBO algorithm then iterates in three steps: 1) update the posterior expectation of f using the learned GP model for new observed values (y, θ), 2) build an acquisition or utility function $g(\theta)$ that measures how desirable a certain θ is to maximize f, and 3) select the parameter setting $\hat{\theta} = \text{argmax}_{\theta \in \Theta} g(\theta)$ that maximizes the acquisition function and defines $(f(\theta), \theta)$ as a new observed value.

B. Feedback Control Using Time-Bounded Sequential Parameter Optimization

In this section, we extend the time-bounded sequential parameter optimization (SPO) algorithm in [29] to tune the parameter settings of different data-processing operators in a computer vision pipeline. Given an input α and its corresponding computed output result π, a performance metric $f(\theta, \pi)$ is defined. This metric describes the quality of a result π computed by the pipeline data-processing operators with input parameter setting θ.

Given the pipeline h, we can define the following feedback loop using a Cut operator on the output stream I_k, where $I'_k, R \triangleq \text{CUT}()(I_k)$:

$$\Big(\text{LEFT-MULT}() \circ h \circ \text{CUT}() \circ (\mathsf{I}_1 \mathbin{+\!\!+} \text{REDUCE}(u, g_{spo}))\Big) \uparrow^1 . \tag{16}$$

The stream R is forwarded to a Reduce operator, which applies a time-bounded SPO function $g_{spo} : U \times X \rightarrow U \times Y$ to produce a feedback stream F. This stream contains that includes the candidate input parameter settings for the controlled operators of h. Using a LeftMult operator, each parameter setting in the feedback stream $\theta \in F$ is attached to a corresponding input instance $\alpha \in I_0$ to produce the merged stream I_m of pairs (α, θ). Each instance $\alpha \in (\alpha, \theta) \in I_m$ is processed by the pipeline data-processing operators $h_1, ..., h_k$ using the attached parameter setting θ to produce the output $(\pi, \theta) \in I_k$.

The function g_{spo} is defined using Algorithm 1, which extends [29] for parameter tuning of computer vision pipelines. The algorithm takes as input two vectors u and x, and produces two outputs u' and y. The vector u keeps track of the algorithm state through sequential runs. The input x is a sample from the pipeline output stream, and the output y is the output parameter setting that will be applied to the next input instance of the pipeline input stream. The state vector u has several components. The $u.W$ component defines the parameter matrix $u.W = [\theta_1, ..., \theta_n]$. The $u.Y$, and $u.N$ components define the performance vector y, where $y_i = Y[i]/N[i]$ is the average performance of the parameter θ_i, and $1 \leq i \leq n$. $u.N[i]$ tracks the number of input instances used for evaluating the parameter setting θ_i, and $u.Y[i]$ is the sum of the pipeline computed performance over all evaluations of θ_i. The components $u.t_{max}$ and $u.n_{max}$ define the maximum computational time and the maximum number of input instances allowed for evaluating any parameter setting. t_{max} allows us to ignore early parameter settings resulting in high computational time, whereas n_{max} ensures that parameter settings in $u.W$ are fairly compared on a similar number of evaluations. The component $u.T$ accumulates the computational time taken for evaluating parameter settings on input instances. The $u.\theta_o$ component refers to the good parameter setting found so far, and is continuously updated after each call to Algorithm 1. The $u.f$ component defines the metric function. The u.toggle continuously switches between two actions: 1) fit a GP model using the updated state u' and find θ that maximizes the expected improvement [29], and 2) randomly sample a new parameter setting from the parameter space Θ.

Algorithm 1 Time-bounded SPO

Require: $u = (T, W, Y, N, \text{toggle}, \theta_o, t_{max}, n_{max}, f), x = (\pi, \theta, \Delta t)$
Ensure: $u' = (T', W', Y', N', \text{toggle}, \theta'_o, t_{max}, n_{max}, f)$ and $y = \theta_n$
1: **if** $x \neq null$ **then**
2: $i_* = \text{GetIndex}(x.\theta, u.W)$;
3: $u.Y[i_*] = u.Y[i_*] + u.f(x.\pi, x.\theta)$;
4: $u.N[i_*] = u.N[i_*] + 1$;
5: $u.T[i_*] = u.T[i_*] + x.\Delta t$;
6: **end if**
7: $u' = u$;
8: $S = \{i | u'.T'[i] \leq t_{max}$ and $u'.N'[i] \leq n_{max}\}$;
9: **if** $S \neq \varnothing$ **then**
10: $i_r = \text{RandomSample}(S)$;
11: $\theta_n = W[i_r]$;
12: **else**
13: $u' = \text{RemoveWorst}(u')$;
14: **if** u'.toggle **then**
15: $\mathcal{M} = \text{FitModel}(u)$;
16: $\theta_n = \text{SelectConfiguration}(\mathcal{M})$;
17: **else**
18: $\theta_n = \text{RandomSample}(\Theta)$;
19: **end if**
20: $u' = \text{Append}(u', \theta_n)$;
21: $u'.\text{toggle} = !u'.\text{toggle}$;
22: **end if**
23: $y = \theta_n$;
24: $S' = \{i | u'.T'[i] \leq t_{max}$ and $u'.N'[i] = n_{max}\}$;
25: **if** $S' \neq \varnothing$ **then**
26: $i_o = \text{argmax}_{i \in S'}(u'.Y'[i]/u'.N'[i])$;
27: $u'.\theta'_o = u'.W'[i_o]$;
28: **end if**

Initially, Algorithm 1 receives a state vector u initialized with a set of n randomly sampled parameter settings from the parameter space Θ. The algorithm starts by testing whether there is an input sample x. If one exists, then the algorithm finds the index i_* of the parameter setting $x.\theta$ (used to evaluate the output sample $x.\pi$) from the state parameter matrix $u.W$. Next, the algorithm calculates the performance of the parameter setting $x.\theta$ using the metric function f. The measured performance value is accumulated on $u.Y[i_*]$, and $u.N[i_*]$ is incremented. In addition, the computational time $u.\Delta t$ taken by the pipeline in computing the output $x.\pi$ is accumulated to $u.T[i_*]$. After updating the state u, it is assigned to the output state u'. In Step 8, the algorithm locates the set S containing all indexes i of parameter settings with accumulated computational times $u'.T'[i] \leq t_{max}$ and number of evaluations $u'.N'[i] \leq n_{max}$. If the set S is not empty, the algorithm randomly picks a parameter setting from $u.X$ and assigns it to output y for the next evaluation. If S is empty, then all parameter settings in $u.X$ met one or both maximum budgets n_{max} and t_{max}. In this case, the algorithm executes the RemoveWorst function that deletes the worst performing parameter setting θ_j from state u', where

j = argmin$_{1<j<n}(u'.Y'[j]/u'.N'[j])$. Then, a new parameter setting is either randomly sampled from space Θ or found from a fitted GP model by maximizing the expected improvement. The u'.toggle variable controls the decision. The new setting is then added to the state u' using the append function and the u'.toggle variable is inverted. Finally, Steps 23 to 26 set the output parameter setting that will be evaluated next, and find a good parameter setting among the settings that met $u.n_{max}$ and with time budget $\leq t_{max}$.

VII. Experimental Results

In this section, we present an experimental evaluation of the time-bounded SPO algorithm, Algorithm 1. For simplicity and without loss of generality, we use the streaming pipeline of the road-boundary detection algorithm as our case study (see Section V-B).

A. Case Study

We can define a feedback loop for the road-boundary detection pipeline by setting $h = h_8$ in Equation 16, and h_8 is given by Equation 14. In this case, a feedback branch is formed by applying the Cut operator to sample the output stream Y and create the return stream R. This stream is used as input to the Reduce operator that executes the function g_{spo}, which applies Algorithm 1 for sequential parameter optimization. The Reduce operator produces the feedback stream F, which is merged with the input video stream V using the Left-Mult operator.

For the road-boundary detection algorithm, the following numerical parameter vector is identified: $\theta = (N, L, \lambda, \tau, \epsilon)$, where N is the number of superpixels, L is the line segment length used in the polygon approximation step, λ is the decay rate of online clustering, τ is the ranking threshold used to select the top-ranked clusters after confidence assignment, and finally ϵ is the parameter used by [28] for calculating cluster confidence. Each parameter has a range of values defined by a lower and upper limit. The limits define the parameter space Θ and are specified as follows: $N \in [25, 150]$; $L \in [5, 30]$; $\lambda \in [0.1, 0.6]$; $\tau \in [0.1, 0.6]$; and $\epsilon \in [5 \times 10^{-3}, 5 \times 10^{-2}]$.

B. Experimental Evaluation

We perform experiments on the two datasets used by [28] for performance evaluation. Dataset 1 contains 50 short low-resolution video sequences collected from 14 different camera locations at a resolution of 320 × 240. Dataset 2 is a single long video sequence of 1,627 frames recorded at 704 × 480 resolution and has the camera changing its viewing directions to focus on different regions in the traffic scene. Both datasets have ground-truth identifying road regions. In [28], all results were computed using the following parameter vector $\theta = (100, 10, 0.2, 0.2, 0.005)$. This vector was found by creating a grid that divides the range of each parameter into five intervals (using the limits defined in the previous section) and randomly selecting parameter settings from the grid. A manual search is applied to select the setting that results in a good accuracy.

To evaluate the effectiveness of the time-bounded SPO algorithm in tuning the parameters of the case study, we compare the quality of the good parameter setting found using Algorithm 1 to the one selected by manual search. The comparison is performed on the long video stream of Dataset 2, where the first 814 frames of the dataset are used for training the GP model, and the remaining 813 frames are used for testing. The quality measure for a parameter setting θ is defined as follows:

$$Q(\theta) = \sum_{i=1}^{n} \frac{\delta_{\theta i}(P_{\theta i} + R_{\theta i})}{2n}, \quad (17)$$

where $n = 814$ is the number of training frames, $\delta_{\theta i} \in (0, 1)$ is a Boolean determining whether or not frame i was used to evaluate the setting θ. In addition, $P_{\theta i}$ and $R_{\theta i}$ are the precision and recall of detected road regions computed for frame i using the parameter setting θ. The precision is measured as the ratio between the area of intersection of the detected road boundary and ground truth to the area of the detected road boundary. The recall is the ratio between the area of intersection between the detected road boundary and the ground truth to the area of the ground truth.

Figure 4 shows the performance curves of the time-bounded SPO algorithm in selecting good parameter settings over time and on the training dataset. Figure 4(left) shows the measured recall value of every good parameter setting found over time. Figure 4(right) shows a similar plot for precision values. The SPO algorithm is executed for 11 minutes. It took around 2 minutes to converge to a good parameter setting with a precision of 93% and a recall of 73%. The tuning algorithm then selected parameter settings with nearly similar performance for the remaining time.

Table IV compares the results between the statistical road boundary detection method of [28], the Gabor based road boundary detection method of [48], and the convolutional neural network (CNN) based road boundary detection method of [49] against the [28]+SPO method. The results of [28]+SPO are reported on the testing dataset. These results show that both the precision and recall of [28] are improved using the time-bounded SPO algorithm. The [28]+SPO pipelined method achieves an average precision of 92% and an average recall of 72%, whereas the pipelined method of [28] achieves 90% and 71%, respectively. The runtime is the same for [28]+SPO as in [28]. This is because the feedback loop defined by the pipeline of [28]+SPO uses the Cut operator for sampling the output stream, which decouples the output stream from the feedback stream.

C. Throughput versus latency analysis

The key performance metrics for streaming workflows are throughput and latency [50], [51], [52]. The throughput measures the rate at which data tuples enter or exit a system. Equivalently, the period is the inverse of throughput, and it measures the interval between the system entry times of two consecutive tuples. The latency is the interval between the system entry and exit times for a given tuple, so it measures the overall response time of the system in processing the tuple. Data tuples may have different latencies; hence, the maximum response time defines the system latency. Although a stream algebra can mathematically define a given streaming workflow, the aim is to construct a mathematical expression that provides an efficient execution plan. So, it is naturally desirable to define mathematical expressions that minimize latency and maximize throughput; however, the two criteria are opposite to each other, and one should find a good trade-off.

We study the throughput and latency of the streaming pipeline of the road boundary detection algorithm. Throughput is the inverse of the period, which is the slowest operator

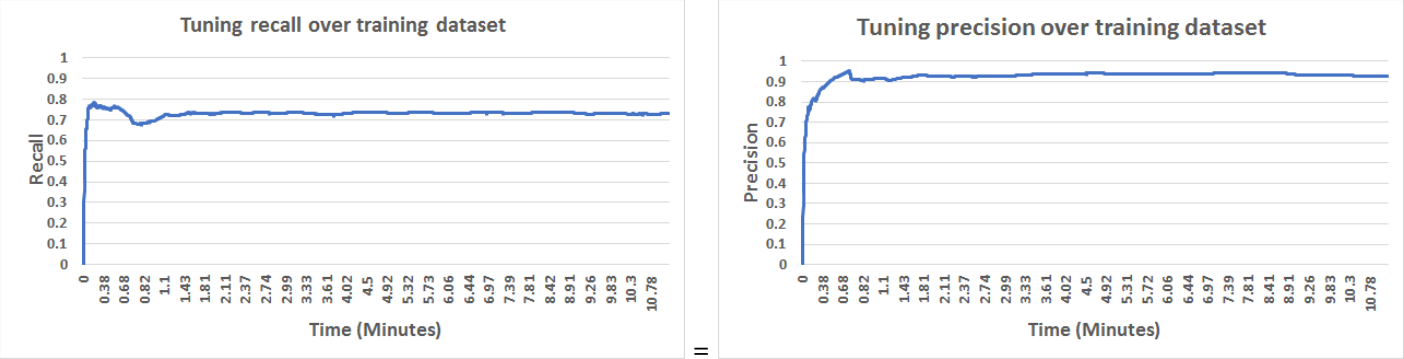

Fig. 4: Using the time-bounded sequential parameter optimization (SPO) algorithm to select parameter settings that maximize precision and recall measures on the training dataset. At each time step, we record the precision and recall of the good parameter setting found so far.

Methods	Precision	Recall	Average Runtime (seconds)
[28] + SPO	$92\%_{\pm 20\%}$	$72\%_{\pm 19\%}$	0.13
Statistical [28]	$90\%_{\pm 11\%}$	$71\%_{\pm 20\%}$	0.13
Gabor [48]	$78\%_{\pm 14\%}$	$43\%_{\pm 12\%}$	54.9
CN24 [49]	$64\%_{\pm 26\%}$	$49\%_{\pm 43\%}$	81.5

TABLE IV: Comparing the results of the statistical road boundary detection method of [28] method before and after applying the sequential parameter optimization algorithm on Dataset 2 and using the mean and standard deviation statistics for both precision and recall. The Gabor [48], and CN24 [49] methods are also included in the comparison. The last columns show average runtime in seconds for each method.

computation or communication time in the pipeline. Figure 3 shows the algebraic description of this pipeline. We tested the pipeline on a multi-core machine with data communicated between the concurrent operators using pointers, so the communication cost between operators are ignored. Given the pipeline in Figure 3, The boundary detection branch starting with stream V_2 and ending with stream B has a latency of 0.049 seconds and a period of 0.012 seconds, on the low-resolution images of Dataset 1, whereas for the standard resolution images of Dataset 2, these numbers are 0.125 and 0.03 seconds, respectively. The activity detection branch starting with stream V_3 and ending with stream A has a latency of 0.013 seconds and a period of 0.01 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.032 and 0.025 seconds, respectively. The bottom branch starting with stream V_1 and ending with output stream Y has a latency of 0.002 seconds and a period of 0.001 seconds, on Dataset 1, whereas for Dataset 2, these numbers are 0.004 and 0.002 seconds, respectively. Notice that the activity detection branch is decoupled from the other branches. The boundary detection and the bottom branches are synchronized together by the Copy operator. So, the overall period and latency are given by the slowest branch, which is the boundary detection branch in this case. The overall latency and period are 0.049 and 0.012 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.125 and 0.03 seconds, respectively. If Copy is replaced by Cut with V' stream as the asynchronous output, then the boundary detection branch is decoupled from the bottom branch. In this case, the overall latency and period are given by the bottom branch. The overall latency and period become 0.002 and 0.001 seconds on Dataset 1, whereas for Dataset 2, these numbers are 0.004 and 0.002 seconds, respectively. This study shows that our algebra can manipulate the data-flow rates by either decoupling or synchronizing slower and faster sections of streaming pipelines.

D. Feedback-Control for Algorithm Selection

Algorithm selection is a popular problem that appears in a wide range of situations. This problem was originally proposed by [53], and aims to find the best algorithm to solve a given problem. In this section, we show how the general forms of Map and Reduce operators, presented by Definitions 3.3 and 3.4, can enable selecting a good algorithm for a given task. We focus on selecting a good background subtraction algorithm for the Reduce operator extracting scene activity in Equation 11. In this case, we create the function list g_2 with three functions for the following background subtraction algorithms: CodeBook [43], DPAdaptiveMedianBGS [54], and DPWrenGABGS [55]. For each algorithm, we use the default parameter setting defined by [56]. We add the parameter $j \in \{1, 2, 3\}$ to the parameter space Θ, where j defines the index of the algorithm in g_2. A chosen good algorithm then improves the quality function $Q(\theta)$ defined by Equation 17. We then apply algorithm 1 as described in Section VII-B. The results indicate that DPWrenGABGS is a good algorithm for Dataset 1, whereas for Dataset 2, CodeBook is a good algorithm. The accuracy was however found very similar to that of the [28]+SPO in Table IV.

VIII. Comparisons to Other Data-flow Approaches

In this section, we provide comparisons between our formal stream algebra and the vision stream processing platforms of [25] and [26]. The system of [25] represents a class of vision

stream processing systems built on top of Spark streaming and Kafka messaging. A data-flow graph in this system contains a set of vision processing modules implemented using Spark streaming with data communicated between modules using Kafka message queues. Notice that the vision processing modules are defined by users and represent data transformation tasks. The platform of [26] constructs vision processing pipelines using Apache Storm. Here the pipeline is a sequence of Map and Reduce operators implemented as Storm Bolts, which are custom stream transformation functions.

Data-Flow Control. The platforms of [25] and [26] only address synchronized data-flow graphs. In [25], a data transformation operator or module can receive more than one input; however, if these inputs are not received at the same time, they will be cached until all inputs are available for the operator to start execution. This can be viewed in our stream algebra as using a Mult operator to gather inputs and forward them to the data transformation operator. In [26], the data-flow graph is formed as a linear and synchronized Storm pipeline. So there is a lack of flow control operators in [25] and [26], which limits these systems from manipulating the data-flow rates similar to what we did in the case study of Section VII-C.

Feedback Control. The platform of [25] enforces data-flow graphs to be directed acyclic, which limits defining feedback control loops. The system of [26] implements vision processing operators as custom Storm Bolts. This allows users to define feedback loops as custom Bolts; however, the users should manually implement all the primitive operations required for feedback control. Our stream algebra on the other hand provides a formal definition of feedback control, which can simplify the implementation of feedback loops in current systems.

Formal Definitions. The systems of [25] and [26] lack the formal definition of the data-flow pipelines. We showed that the formal definition and theoretical study of our stream algebra allow a formal definition of the computer vision pipelines. In addition, we were able to infer a set of valid axioms in Table III that enables several optimizations to the data-flow graphs. For example, the axioms S1, S2, and S3 allows merging Map and Reduce operators that can eliminate large communication costs between the data stream operators.

We think that implementing the primitives of our stream algebra in existing stream processing systems, such as [25] and [26], resolves their discussed limitations to formally express a wider range of computer vision algorithms.

A. Discussion

We presented several examples that show the ability of our stream algebra to naturally describe several computer vision algorithms. We expressed the vision pipeline of each example using formal algebraic equations that manipulate vision streams. This provides an abstract representation that highlights the semantics of the different algorithmic components of computer vision algorithms and simplifies the process of building scalable computer vision systems.

The stream algebra provides several operators for data stream processing and flow rate control in synchronous and asynchronous data-flow networks. The data-processing operators, which include Map and Reduce, implement data transformations on the input vision streams. The rate-control operators, which include Cut and Latch, manipulate the data-flow rates. Rate-control operators can resolve blocking and slow operations by synchronizing and matching between the different data-flow rates of vision streams, thus supporting real-time streaming. This provides the important advantage of handling unbounded data rates of continuous (and possibly infinite) vision streams.

The theoretical study in Section IV shows that our stream algebra forms a data transformer model of BNA [41], [42], and describes its compositional and constant operators. This allows the composition of mathematical equations that describe data-flow networks using our stream operators and the equational primitives of BNA. Consequently, our stream algebra fulfils the main and additional BNA axioms in Tables I and II. We also extend these axioms by the set of axioms in Table III.

The axioms allow the ability to define equivalence relations between different data-flow networks and simplify complex ones. For example, the axiom S1 uses function composition to combine a sequence of Map operators into a single Map operator. Similarly, the axioms S2 and S3 combine Map operators with a Reduce operator. The runtime can dynamically choose between these different implementation plans to maximize performance and reduce latencies required to move data between concurrent stream-processing operators.

Experimental results showed the ability of our stream algebra to naturally describe feedback-control and to implement a general parameter optimization algorithm for performance tuning of streaming computer vision pipelines. The case study showed that the algebraic feedback-control primitives can be combined with the time-bounded SPO algorithm to tune numerical parameters of pipelined computer vision functions.

The case study focused on tuning the streaming pipeline for the road-boundary detection algorithm on a single-input stream (see Figure 3). However, we can scale up the algorithm to process several streams by creating multiple instances of the streaming pipeline, one for each stream. In this case, the visual content of each stream will guide the parameter optimization of the dedicated processing pipeline of the stream. Thus, learning a different parameter setting for each stream, which is adapted to the lighting, structure, and environmental conditions presented in the stream. However, this requires obtaining a training dataset for each stream. These datasets can be obtained using a human operator that labels the road boundaries in one or more sequences of stream images.

The algorithm selection experiment in Section VII-D shows that we can optimize the pipeline performance over Map and Reduce functions, which can be different algorithms performing the same task but with different accuracy and runtime profiles. Although we presented algorithm selection for only one operator, we can scale up algorithm selection to all Map and Reduce operators in a large-scale pipeline. In this case, we need common methods for automatically constructing the space of all user-defined functions and their parameters. This can be efficiently achieved by using the fact that the Map and Reduce operators are aware of the number of functions and the parameters of every function. This information is automatically added to the header section of data tuples, and can be read by a feedback loop to optimize over the parameter space of all user-defined functions. This enables dynamic reconfiguration by allowing the data-processing operators to switch between different functions at runtime. This is very important in large-scale systems processing a large number of incoming streams with different data rates. An incoming stream may have its data-flow rate change at a much faster

rate. In this case, the pipeline may decide to switch the current processing functions to faster functions to match the new incoming data-flow rate. This decision can be performed dynamically using the feedback-control mechanisms of our stream algebra. Therefore, our algebra opens a new research direction in enabling dynamic reconfiguration in large-scale computer vision pipelines.

The time-bounded SPO algorithm, Algorithm 1, is applied in the case study as an offline learning approach. In this case, the algorithm finds a good parameter setting using the given training and testing datasets. Then, the good parameters are used as the default setting for future stream processing. However, online learning is also possible using partially labelled data, where a human operator can manually label some segments of the input stream. These segments can be then used to further optimize and adapt pipeline parameters against previously unseen environmental changes in the traffic scene.

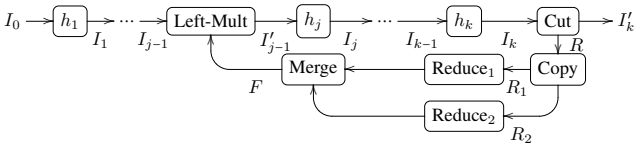

Fig. 5: Example of applying the idea of boosting to merge the output of two different parameter-tuning algorithms and select a good parameter setting.

Although the case study applied Algorithm 1 [29] for parameter tuning, several other model-based [57], [46], [47], [45] and model-free [58], [59], [60], [44], [45], [61] parameter-tuning algorithms can also be applied. The racing algorithm [59], for example, can be used if a list of candidate settings can be predetermined. A good setting can be selected by iteratively evaluating each candidate setting on a stream of input instances. The method of [62] can be used to optimize categorical parameters using decision tree models. Such methods can be applied by replacing the processing function g_{spo} in Equation 16.

Moreover, the feedback-control loop presented in the case study can be extended to multi-loop feedback-control (see Figure 2) for scaling up parameter optimization. In this case, each feedback loop can have the Reduce operator applying similar or different parameter-tuning methods. By treating these methods as weak learners for estimating the performance response surface of the quality metric (see Equation 17), we can apply the idea of boosting from machine learning [63]. Boosting combines a set of weak learners to produce a single strong learner. Assuming m weak learners, the Merge operator (see Figure 5) can find a good parameter setting θ_* from the output estimates $\pi = \{\theta_1, ..., \theta_m\}$ of weak learners by applying the select function $\theta_* = \text{argmax}_{\theta \in \pi}\{Q(\theta)\}$. Here, each learner produces a pair $(\theta, Q(\theta))$.

The case study, experimental results, and discussion show the effectiveness of using our stream algebra for general parameter tuning of the computer vision pipeline. We think that the ability of our stream algebra to flexibly apply and scale up parameter tuning will open several future opportunities in building and optimizing large-scale streaming computer vision systems.

IX. Conclusion

In this work, we addressed existing challenges in building large-scale computer vision systems processing image and video streams. These challenges include the lack of formal and scalable frameworks for building and optimizing streaming computer vision pipelines. Additionally, many existing computer vision algorithms are computationally expensive and cannot efficiently scale up for processing large-scale data. As a step toward in overcoming these challenges, we presented formal methods for building scalable computer vision systems.

We developed a stream-algebra framework for mathematically expressing computer vision pipelines (online vision systems that process image and video streams). The algebra defines an abstract set of concurrent operators with formal semantics that manipulate image and video streams. The algebra provides operators for both data processing and rate control. The data-processing operators perform data transformations on input image and video streams, whereas the rate-control operators allow decoupling and synchronizing between the data-flow rates of different computer vision pipelines, thus supporting seamless integration between different computer vision tasks to build large-scale systems. The algebra also can naturally express feedback-control loops, thus enabling optimization tasks, such as parameter tuning and iterative optimization.

Using a case study, we showed that the feedback-control definitions of our stream algebra can implement a general parameter optimization algorithm for parameter tuning of streaming computer vision pipelines. Experimental results showed the effectiveness of combining the feedback-control primitives of our algebra with a general sequential parameter optimization algorithm as a common optimization method for parameter tuning in large-scale streaming pipelines. Currently, we are extending Kafka streams by implementing an API for the stream algebra that will enable computer vision researchers to easily develop and build large scale computer vision systems. In future, we are planning to address the new research directions open by the algebra for parameter tuning, iterative optimization, dynamic reconfiguration, and performance tuning of large scale computer vision systems.

References

[1] B. Zhao, L. Fei-Fei, and E. Xing, "Online detection of unusual events in videos via dynamic sparse coding," in *CVPR*, Colorado Springs, June 2011, pp. 3313–3320.
[2] A. Meghdadi and P. Irani, "Interactive exploration of surveillance video through action shot summarization and trajectory visualization," *IEEE Trans. on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2119–2128, Dec 2013.
[3] S. Yenikaya, G. Yenikaya, and E. Düven, "Keeping the vehicle on the road: A survey on on-road lane detection systems," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 2:1–2:43, Jul. 2013.
[4] C. Loy, T. Hospedales, T. Xiang, and S. Gong, "Stream-based joint exploration-exploitation active learning," in *CVPR*, 2012, pp. 1560–1567.
[5] M. S. Ryoo, "Human activity prediction: Early recognition of ongoing activities from streaming videos," in *ICCV*, Barcelona, Spain, 2011, pp. 1036–1043.
[6] C. Lu, J. Shi, and J. Jia, "Online robust dictionary learning," in *IEEE CVPR*, 2013, pp. 415–422.
[7] C. Xuand, C. Xiong, and J. Corso, "Streaming hierarchical video segmentation," in *ECCV*, vol. VI, 2012, pp. 626–639.
[8] N. Harbi and Y. Gotoh, "Spatio-temporal human body segmentation from video stream," in *Computer Analysis of Images and Patterns*. Springer, 2013, vol. 8047, pp. 78–85.
[9] J. Yang, J. Luo, J. Yu, and T. Huang, "Photo stream alignment and summarization for collaborative photo collection and sharing," *IEEE Trans. on Multimedia*, vol. 14, no. 6, pp. 1642–1651, 2012.

[10] G. Kim and E. Xing, "Jointly aligning and segmenting multiple web photo streams for the inference of collective photo storylines," in *CVPR*, 2013, pp. 620–627.

[11] M. A. Helala, K. Q. Pu, and F. Z. Qureshi, "Towards efficient feedback control in streaming computer vision pipelines," in *ACCV Workshops*, 2015, pp. 314–329.

[12] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, vol. 29, 2003, pp. 81–92.

[13] A. Bifet, G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl, "Moa: Massive online analysis, a framework for stream classification and clustering," in *JMLR*, 2010, pp. 44–50.

[14] D. Wang, E. A. Rundensteiner, and T. R. I. Ellison, "Active complex event processing over event streams," *VLDB*, vol. 4, no. 10, pp. 634–645, Jul. 2011.

[15] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom, "Characterizing memory requirements for queries over continuous data streams," Stanford InfoLab, Technical Report 2002-29, May 2002.

[16] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *SIGMOD*, New York, NY, USA, 2013, pp. 577–588.

[17] G. Chkodrov, P. Ringseth, T. Tarnavski, A. Shen, R. Barga, and J. Goldstein, "Implementation of stream algebra over class instances, Google patents," Patent US20 130 014 094 A1, jan, 2013.

[18] M. Broy and G. Stefanescu, "The algebra of stream processing functions," *Theoretical Computer Science*, vol. 258, no. 1-2, pp. 99 – 129, 2001.

[19] J. Carlson and B. Lisper, "An event detection algebra for reactive systems," in *Proceedings of the 4th ACM International Conference on Embedded Software*, 2004, pp. 147–154.

[20] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "A general algebra and implementation for monitoring event streams," Cornell University, Technical Report, 2005.

[21] Apache Storm, http://storm.apache.org/, accessed: 2016-07-17.

[22] A. Kinesis, aws.amazon.com/kinesis/, accessed: 2014-02-27.

[23] Apache Kafka, http://kafka.apache.org, accessed: 2016-07-17.

[24] Apache Spark, http://Spark.apache.org, accessed: 2016-07-17.

[25] K. Yu, Y. Zhou, D. Li, Z. Zhang, and K. Huang, "A large-scale distributed video parsing and evaluation platform," in *Intelligent Visual Surveillance*. Singapore: Springer Singapore, 2016, pp. 37–43.

[26] D. Tabernik, L. Čehovin, M. Kristan, M. Boben, and A. Leonardis, "A web-service for object detection using hierarchical models," in *Computer Vision Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 93–102.

[27] H. Zhang, J. Yan, and Y. Kou, "Efficient online surveillance video processing based on spark framework," in *Big Data Computing and Communications*. Springer International Publishing, 2016, pp. 309–318.

[28] M. A. Helala, F. Z. Qureshi, and K. Q. Pu, "Automatic parsing of lane and road boundaries in challenging traffic scenes," *Journal of Electronic Imaging*, vol. 24, no. 5, p. 053020, 2015.

[29] F. Hutter, H. Hoos, K. Leyton-Brown, and K. Murphy, *Time-Bounded Sequential Parameter Optimization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 281–298.

[30] R. Schuster, R. Mörzinger, W. Haas, H. Grabner, and L. V. Gool, "Real-time detection of unusual regions in image streams," in *Proceedings of the International Conference on Multimedia*, 2010, pp. 1307–1310.

[31] Y. Cao, D. Barrett, A. Barbu, S. Narayanaswamy, H. Yu, A. Michaux, Y. Lin, S. Dickinson, J. Siskind, and S. Wang, "Recognize human activities from partially observed videos," in *CVPR*, Oregon, USA, June 2013, pp. 2658–2665.

[32] T. Marlin, *Process Control: Designing Processes and Control Systems for Dynamic Performance*, ser. Chemical Engineering Series. McGraw-Hill, 1995.

[33] P. Kisilev and D. Freedman, "Parameter tuning by pairwise preferences," in *BMVC*, 2010.

[34] J. Sherrah, "Learning to adapt: A method for automatic tuning of algorithm parameters," in *ACIVS*, 2010, pp. 414–425.

[35] J. S. III and D. Ramanan, "Self-paced learning for long-term tracking," in *CVPR*, Washington, DC, USA, 2013, pp. 2379–2386.

[36] D. Chau, J. Badie, F. Bremond, and M. Thonnat, "Online tracking parameter adaptation based on evaluation," in *IEEE International Conference on AVSS*, Aug 2013, pp. 189–194.

[37] M. A. Helala, K. Q. Pu, and F. Z. Qureshi, "A stream algebra for computer vision pipelines," in *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 800–807.

[38] ——, "A formal algebra implementation for distributed image and video stream processing," in *Proceedings of ICDSC*, 2016, pp. 84–91.

[39] Z. Lai, Q. Luo, and X. Jia, "Revisiting multi-pass scatter and gather on gpus," in *ICPP*, no. 18, 2018, pp. 1–11.

[40] D. Kim, V. Lee, and Y. Chen, "Image processing on multicore x86 architectures," *IEEE Signal Processing Magazine*, vol. 27, pp. 97 – 107, Apr 2010.

[41] J. A. Bergstra and G. Stefanescu, "Network algebra for synchronous and asynchronous dataflow," *Logic Group Preprint Series, Utrecht Research Institute for Philosophy*, vol. 122, 12 1994.

[42] J. Bergstra, C. Middelburg, and G. Stefanescu, "Network algebra for synchronous dataflow," *International Journal of Computer Mathematics*, vol. 65, 03 2013.

[43] K. Kim, T. Chalidabhongse, D. Harwood, and L. Davis, "Background modeling and subtraction by codebook construction," in *ICIP*, vol. 5, Oct 2004, pp. 3061–3064.

[44] F. Hutter, H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, 2007, pp. 1152–1157.

[45] F. Hutter, H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, Sep. 2009.

[46] F. Hutter, H. Hoos, and K. Leyton-Brown, "An evaluation of sequential model-based optimization for expensive blackbox functions," in *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '13 Companion. New York, NY, USA: ACM, 2013, pp. 1209–1216.

[47] T. Bartz-Beielstein, C. W. G. Lasarczyk, and M. Preuss, "Sequential parameter optimization," in *IEEE Congress on Evolutionary Computation*, vol. 1, Sept 2005, pp. 773–780 Vol.1.

[48] H. Kong, J. Audibert, and J. Ponce, "General road detection from a single image," *IEEE Transactions on Image Processing*, vol. 19, no. 8, pp. 2211–2220, Aug 2010.

[49] C. Brust, S. Sickert, M. Simon, E. Rodner, and J. Denzler, "Convolutional patch networks with spatial prior for road detection and urban scene understanding," in *Proc. of 10th International Conference on Computer Vision Theory and Applications*, March 2015, pp. 11–14.

[50] A. Benoit and Y. Robert, "Complexity results for throughput and latency optimization of replicated and data-parallel workflows," *Algorithmica*, vol. 57, no. 4, pp. 689–724, 2008.

[51] A. Benoit, H. Kosch, V. Rehn-Sonigo, and Y. Robert, "Multi-criteria scheduling of pipeline workflows (and application to the jpeg encoder)," *International Journal of High Performance Computing Applications*, vol. 23, no. 2, pp. 171¡V–187, 2009.

[52] A. Benoit, Ümit V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling: Models and algorithms," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 50:1–50:36, August 2013.

[53] J. R. Rice, "The algorithm selection problem," ser. Advances in Computers. Elsevier, 1976, vol. 15, pp. 65 – 118.

[54] N. McFarlane and P. Schofield, "Segmentation and tracking of piglets in images," *Machine Vision and Applications*, vol. 8, no. 3, pp. 187–193, 1995.

[55] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland, "Pfinder: real-time tracking of the human body," *IEEE PAMI*, vol. 19, no. 7, pp. 780–785, 1997.

[56] A. Sobral and A. Vacavant, "A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos," *Computer Vision and Image Understanding*, vol. 122, pp. 4 – 21, 2014.

[57] D. Jones, M. Schonlau, and W. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, Dec 1998.

[58] B. Adenso-Diaz and M. Laguna, "Fine-tuning of algorithms using fractional experimental designs and local search," *Operational Research*, vol. 54, no. 1, pp. 99–114, Jan. 2006.

[59] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, *F-Race and Iterated F-Race: An Overview*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 311–336.

[60] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '02, San Francisco, CA, USA, 2002, pp. 11–18.

[61] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, 2009, pp. 142–157.

[62] F. Hutter, H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, ser. LION'05, 2011, pp. 507–523.

[63] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*, 1st ed. Chapman & Hall/CRC, 2012.

Mohamed A. Helala received the B.Sc. and M.Sc. degrees in Computer Engineering from Benha University, Egypt, in 2004 and 2010, respectively, and the Ph.D. degree in Computer Science from the University of Ontario Institute of Technology (UOIT), Oshawa, Canada, in 2018. He is a Big Data Consultant working on building and optimizing large scale data processing systems. His research interests include computer vision, machine learning, data stream processing, and big data processing systems.

Faisal Z. Qureshi (Senior Member IEEE, Member ACM and Secretary and Member CIPPRS) received the B.Sc. degree in Mathematics and Physics from Punjab University, Lahore, Pakistan, in 1993, the M.Sc. degree in Electronics from Quaid-e-Azam University, Islamabad, Pakistan, in 1995, and the M.Sc. and Ph.D. degrees in Computer Science from the University of Toronto, Toronto, Canada, in 2000 and 2007, respectively.

He is a Professor of Computer Science in the Faculty of Science, Ontario Tech University, where he leads the Visual Computing Lab. His research focuses on computer vision, and his scientific and engineering interests center on the study of computational models of visual perception to support autonomous, purposeful behavior in the context of ad hoc networks of smart cameras. Dr. Qureshi is also active in journal special issues and conference organizations. He served as the general co-chair for the Workshop on Camera Networks and Wide-Area Scene Analysis (co-located with CVPR) in 2011-13. He also served as the co-chair of Computer and Robot Vision (CRV) conference 2015/16 meetings.

Ken Q. Pu is an Associate Professor of Computer Science at University of Ontario Institute of Technology. He has been working on various aspects of database systems. His research interest is in new data index structures and query processors for novel data-driven applications. Recently, Ken has been working on high performance data stream processors to support real-time computer vision applications.