High Performance Stream Processing and Optimizations

Farley Lai
poyuan-lai@uiowa.edu

July 24, 2015

1 Introduction

Building scalable systems that process streams of data requires developers to take advantage of the parallelism capabilities offered by today's computer architectures. Existing imperative programming languages provide programmers low-level primitives such as threads, locks, and semaphores. However, programs developed using these primitives tend to be plagued by race conditions, deadlocks, hard to understand and debug non-deterministic behaviors **Lee2006** Acknowledging these limitations, some programming language extensions and libraries (e.g., OpenMPI **Gabriel2004** OpenMP **Nc1998**) have been proposed to simplify the programming. Nevertheless, all these options still burden the programmer with annotating parallelism and specifying data sharing attributes to ensure data consistency.

In recent years, dataflows have attracted significant attention as a model for building highly parallel stream processing applications. According to this model, an application is defined as a graph of processing elements that are connected by communication channels. The processing elements may execute in parallel as long as they have sufficient data to process. A key feature of the data flow model is that it explicitly capture parallelism and data dependencies between processing elements.

Even though the data flows provide a simple computational model, using this model to build scalable systems is challenging as naive implementations introduce unexpected run-time scheduling overhead, consume significant memory resources, and are not energy efficient. Consequently, our goal is to develop compiler optimizations and efficient run-time environments for scalable data flow systems. In this report, we trace back to the origin of dataflow models of computation including Kahn process networks (KPNs) **Kahn1974** and synchronous data flow (SDF) **Lee1987** to identify what the desirable properties are that allow for the development of practical stream

processing systems. As it is usually the case, efficiency usually comes at the cost of more rigid computational models. We will explore this in Section 2. To fully exploit the exposed parallelism of stream programs, existing stream processing optimizations may be applied. We survey the techniques and further optimizations on dynamic scheduling, memory layouts and linear analysis in Section 3. Mobile sensing applications (MSAs) utilizing stream processing engines in the backend get popularity especially in the health-care domain on the demand of continuous monitoring. However, limited computing resources and battery life on mobile devices post a direct challenge to every long-running applications that also require high performance real-time processing of sensor data. As a result, several energy saving methodologies are explored with respect to stream processing in Section 4. In view of massive incoming data streams at very high velocity in the cloud computing domain, new challenges to large scale stream processing in a distributed environment and representative applications including Kineograph **Cheng2012** Naiad **Murray2013** are covered in Section 5, plus the evaluation of synchronous and asynchronous execution modes. Section 6 concludes and identifies areas for future work..

2 Models of Computation

A model of computation (MoC) formally specifies the behaviors and properties of a system for developers to make sensible design decisions. In the following, we will explore different models of computations that execute data flows in parallel but have different characteristics in terms of determinacy, expressivity, and memory requirements. Determinate systems guarantee the order of output given fixed input regardless of the parallel scheduling. Expressivity shows the class of programs the language or model can express and may raise concerns about non-determinism in the system and termination of executions. The memory requirements may be bounded or unbounded and system designers have to ensure the implementation consumes no more than available resources.

The models of computation for stream processing can be traced back to the dataflow architectures in 1970s. In contrast with the control flow based von Neumann architecture where the execution of a program compiled as a sequential instruction stream follows a program counter, a dataflow program is modeled as a directed graph that has functions as nodes and communication channels as edges. The functions can be invoked simultaneously given the availability of required inputs. The graph explicitly exposes the parallelism of a program explicitly, thus simplifying program analysis and optimizations. In the following, we will present two representative dataflow models: Kahn Process Networks (KPNs) **Kahn1974** and Synchronous Data Flow (SDF) **Lee1987** The two models differ in their properties which affect our ability to build

scalable stream processing systems.

2.1 Kanh Processes Networks

The requirements of KPNs are simple but effective to guarantee desirable properties such as determinacy for parallel programming. A KPN is a network of processes connected by possibly unbounded FIFO channels. No channels are shared across processes. A process always succeeds writing to an output channel but may block waiting on an empty input channel.

A FIFO channel has a sequence of tokens, each of which represents one or more consecutive samples in the channel. A process may have multiple input and output channels. An input sequence from *multiple* input channels is represented as a sequence of tuples, each tuple containing a token per channel. The output is modeled similarly. In KPNs, a process may consume and produce an arbitrary number of tokens from and to its input and output channels.

Each process is a *continuous* function which maps a possibly infinite input sequence of tuples to an output sequence of tuples. We note that KPNs consider continuous functions defined on ordered sets rather than the special case of real numbers. To formally define the *continuity* of processes, an input or output sequence is represented an increasing *chain* of sequences $\chi = \{X_0, X_1, \ldots, X_n\}$ that captures all the possible input or output sequences in an ordered set. The variable X_i is a sequence of i tuples. Accordingly, X_0 is an empty sequence denoted by $\perp$ and X_n is an infinite sequence as $n \to \infty$. A *source* process without input is modeled as consuming an empty sequence while a *sink* process without output produces an empty sequence. χ forms a prefix ordering $\sqsubseteq$ where X_i is the prefix of X_j if $i \leq j$, i.e., $X_0 \sqsubseteq X_1 \sqsubseteq \ldots. \sqsubseteq X_n$. The property holds because channels are FIFO, i.e., χ is a complete partial order (cpo) if and only if for all subsets in χ, there exists an upper bound in χ denoted by $\sqcup\chi$ such that each element in the subset is a prefix of the upper bound. To be specific, X_n is the upper bound in χ and $\sqcap\chi$ is the empty sequence $\perp$ which serves as the prefix of any sequence and is the lower bound of χ.

A process in KPNs is a continous function $F : \chi \to \Psi$ that maps an input sequence in the increasing chain χ to an output seuqecne in another increasing chain Ψ such that the following property holds:

$$F(\sqcup\chi) = \sqcup F(\chi)$$

Informally, it says the upper bound of input to the function is the upper bound of output of the function. This implies the function is order-preserving, namely, *monotonic*. A function F is monotonic if given $X_i \sqsubseteq X_j$, $F(X_i) \sqsubseteq F(X_j)$ holds. It is straightforward to verify that $F(\sqcup\{X_i, X_j\}) = F(X_j) = \sqcup\{F(X_i), F(X_j)\}$ implies

$F(X_i) \sqsubseteq F(X_j)$. Therefore a continuous function must be monotonic. The intuition behind monotonicity allows the process in KPNs to consume and produce sequences of tokens from and to channels incrementally. Support an input sequence $X = x_1.x_2$ such that x_1 is the prefix of X. $F(x_1) \sqsubseteq F(X)$ by continuity implies the consumer process of F is able to incrementally process $F(x_1)$ and then the remaining output induced by x_2.

However, a monotonic function may not be continuous. In this case, it is possible for a process to wait for an *infinite* input sequence before producing any output. Consider the monotonic function F defined in Eq.(**??**) and the increasing input chain χ. In this case, $\sqcup\chi = \lim_{n\to\infty} X_n$ is infinite so $F(\sqcup\chi)=(v)$. Unfortunately, $\sqcup F(\chi)$ can only produce $\perp$ because the process needs to block reading indefinitely to determine if the input is an infinite sequence. This behavior may cause non-determinism and justifies why KPNs require processes to be continuous functions. In practice, simply view processes in a KPN as monotonic since it is rare to have processes which need to determine whether the input is infinite.

$$F(X) = \begin{cases} \perp; \text{ if } X \text{ is finite} \\ (v); \text{ for some integer } v \text{ if } X \text{ is infinite} \end{cases} \quad (1)$$

Determinacy. An important property of KPNs known as the Kahn principle is that KPNs are *determinate* iff given the input and internal sequences, the final output sequences are uniquely determined regardless of the process scheduling policies. This principle holds because all the processes are continuous functions on cpos and by induction, a finite composition of processes is also a continuous function. By the fixed-point theorem, there exists a least fixed point solution to the system of equations formulated from the functions in the KPN. To derive such a solution, start with $\perp$ as initial input sequences and iterate until the output sequence does not change. Specifically, the least fixed solution is the least upper bound on the cpo. Nevertheless, it is still possible to violate the semantics of KPNs without caution and cause non-determinism since dataflow models may be implemented in an imperative host language that allows processes to communicate over shared variables, test if channels are empty as well as call non-deterministic system APIs. A dataflow compiler should alert programmers the potential violation of determinacy.

Boundedness of Channels. Another concern in KPNs is whether channels have bounded capacity. A practical implementation of KPNs would require bounded memory requirements. In the next section, we will describe how we can limit the semantics of the MoC to ensure bounded channels. However, in the following we describe a solution proposed by Parks **Parks1995** proposed a dynamic scheduling approach to guarantee bounded channel buffers. In his approach, the channel capacity begins with at least one and each process would block for writing to a full channel. The scheduler then schedules enabled processes to execute. A process is enabled if its

input is ready as required by KPNs. As expected, the system may reach an artificial deadlock when *all* the processes are blocked, some of which on a full FIFO. In this case, the scheduler increases the lowest full channel capacity until the deadlock is resolved. However, implementations of KPNs based on Parks' approach may result in incomplete output since only global deadlocks are considered. Geilen and Basten **Geilen2003** improved the scheduling despite local deadlock cycles. On the other hand, if unbounded channel capacity is allowed, the fairness of scheduling should be concerned to ensure enabled processes execute infinitely often. Otherwise, unfair scheduling may lead to low throughput even though the output is still determinate.

Expressivity. KPNs have been showed to be Turing-complete in **Buck1993** given boolean tokens, initial tokens in the channels of feedback loops, memoryless processes as well as conditional constructs such as the Switch and Select. A stateful process can be transformed to be memoryless by passing its states through a self-loop channel and reading back in the next invocation. The Switch and Select implement the if-then-else conditionals in common imperative languages.

Though KPNs are powerful in the sense of expressivity, the termination and channel buffer boundedness are undecidable in general. Moreover, the runtime overhead due to the dynamic scheduling could be undesirable for high performance computing. In the next section, we introduce the synchronous data flow which models more restrictive semantics of KPNs but allows to decide the termination and derives static schedules with bounded memory at compile time.

2.2 Synchronous Data Flow

Similar to KPNs, the Synchronous Data Flow (SDF) models a program as a directed graph. However, SFDs require the production and consumption rates of processes during an invocation to be known statically. Process states are maintained through self-loops. The execution flow of the SDF is data-driven: the push model that starts from upstream sources to downstream sinks. There are many derivatives and variants of SDFs (e.g., StreamIt **Thies2002** and Lustre **Halbwachs1991**) that adopt similar semantics for generating static schedules with bounded memory allocations and allocating initial tokens to handle feedback loops. The remainder of the section will cover these aspects of SDFs.

Periodic Static Schedules. Assuming a single processor architecture, it is possible to derive a periodic admissible sequential schedule (PASS) **Lee1987** from a SDF graph and then adapt the schedule for multiprocessor architectures. A PASS is periodic to allow for processing infinite streams of data by executing the program repetitively. It utilizes a finite amount of memory and is sequential for a single processor system. The idea is to simulate process invocations and track the number of tokens in the channel buffer so as to find a schedule such that in the end the buffer

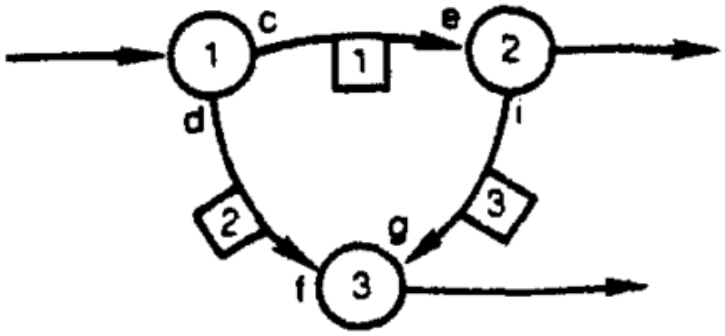


Figure 1: A SDF graph with nodes and edges numbered in order and letter rates.

size in the number of tokens is the same as the beginning of executing the schedule.

To begin with, a *topology matrix* based on a SDF graph with rows representing channels and columns representing processes. An entry in the topology matrix corresponding to the number of tokens consumed or produced by a process. Each positive value indicates that the process produces data while a negative value indicates a process consumes data. Fig. **??** shows a simple SDF with nodes and labeled edges. The topology matrix Γ of the SDF in the figure is given blow

$$\Gamma = \begin{pmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{pmatrix}. \tag{2}$$

To simulate the invocation of a process, Γ is multiplied by an indicator column vector. The vector $v_n(i)$ indicates the process i is executed at time n by setting the i_{th} entry in v_n to one and setting all other entries to zero. In our example, $v_n(i)$ may take the following values:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3}$$

depending on which process is scheduled. A process is *schedulable* if there is sufficient data to consume in its input channel(s). This ensures each time a process in invoked, the corresponding channel buffer never underflows. We track the number of tokens available in a channel using the column vector $b(n)$. The number of tokens in a channel after invoking a process may be computed using the following equation.

$$b(n+1) = b(n) + \Gamma v_n(i)$$

Suppose the initial buffer status is $b(0)$, the simulation iteratively executes processes until the channel buffer status $b(n) = b(0)$ for some $n > 0$. Clearly, the channel buffer

usage is bounded and equal to the maximum number of tokens on each channel during the simulation. Additionally, the sequence of invocations recorded in v_t for $0 \leq t \leq n$ constitutes one feasible PASS.

However, in some cases, the simulation never ends when in no iteration $b(n) = b(0)$. This situation occurs when the rates are inconsistent. It is proved in **Lee1987a** the necessary condition for a SDF graph to have a PASS is

$$\text{rank}(\Gamma) = s - 1,$$

where s is the number of processes. The proof is by induction on the rank of Γ starting with a two-node tree with rank one. Whenever a node and an edge are added to connect the new node, Γ is expanded to have a new column with a nonzero entry in the new row for the new node and edge. In this way, rank(Γ) is always equal to the number of nodes minus one given the SDF graph as a tree. Next, consider the actual SDF graph with more edges which only add rows to Γ without decreasing its rank and thus the following equation holds.

$$s - 1 \leq \text{rank}(\Gamma) \leq s$$

Since the total buffer size change is equal to $\Gamma\Sigma_{t=0}^{n} v_t$ which should be zero for boundedness, the nullity of nonzero $\Sigma_{t=0}^{n} v_t$ can only be one by the rank-nullity theorem and rank(Γ) $= s - 1$ as a result. If rank(Γ) $= s$ for a given SDF graph, any schedule will result in either deadlock or unbounded buffer size. That implies a particular channel has its buffer size either decreasing or increasing unboundedly. After determing the existence of a PASS, it is sufficient to find the schedule by simulating the process invocations. If there is a deadlock before the buffer status is restored, some additional delays are necessary to add to the initial buffer status $b(0)$. This concludes the sufficient condition for the static schedule.

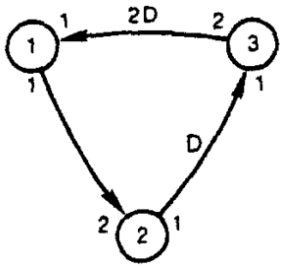


Figure 2: An example SDF graph for constructing a PAPS.

Parallel Schedules. Based on the PASS, constructing a periodic admissible parallel schedule (PAPS) is straightforward. For simplicity, assume the time to execute a

SDF process takes an integral number of time slots. This is realistic given processors with manageable pipelining. Otherwise, some inter-process synchronization may be required. Suppose there are M processors, each of which is assigned a static schedule ψ_i to run such that for each process, the total number of invocations on all processors is a multiple of the number of invocations in a PASS

$$\Sigma\psi_i = J \cdot \phi$$

where ϕ is the corresponding PASS and J is some positive integer called the blocking factor to scale up the PASS for better performance. Fig. **??** gives an example SDF graph with a minimum PASS $\{1, 1, 2, 3\}$ for executing processes 1, 2, and 3 in one period. For $J = 1$ and $J = 2$ and given $M = 2$, two possible PAPS can be constructed as follows:

$$\psi_1 = \{3\}, \psi_2 = \{1, 1, 2\} \text{ for } J = 1$$

$$\psi_1 = \{3, 1, 3\}, \psi_2 = \{1, 1, 2, 1, 2\} \text{ for } J = 2$$

Assume it takes one time slot to execute process 1, two slots for process 2 and three slots for process 3. The makespans on both processors are illustrated in Fig. **??**. Apparently, it is not efficient for $J = 1$ since processor 2 needs to wait for processor 1 to complete (the shadow area) while for $J = 2$, processor 2 is allowed to continue executing and both processors wait no time for each other until the end of one period of the PAPS. Given an appropriate J, finding the optimal partition is a well-known assembly line problem in operations research. Though it is NP-complete **Graham1979** there are efficient approximation algorithms such as **Hu1961** for such constructions.

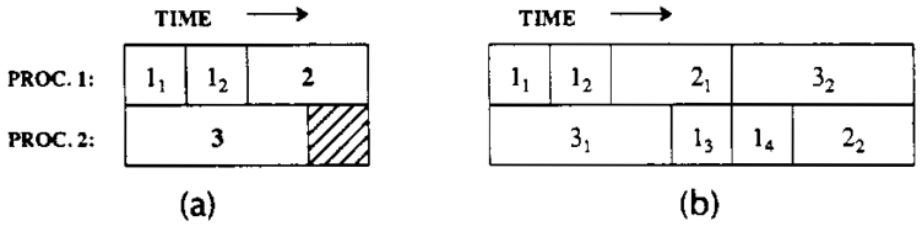


Figure 3: Makespans on two processors for $J = 1$ and $J = 2$ in (a) and (b).

Asynchrony. In some cases, expressing the entire program in a single SDF graph is not feasible due to the existence of asynchronous processes whose production and consumption rates are data dependent, implying the existence of internal *Switch* and *Select* logics which accept control inputs and change the dataflow paths at runtime. Nonetheless, a practical solution is to partition the program into separate SDF subgraphs connected by build-in Switch and Select components which are compiled as

conditional statements on a single processor architecture and inter-process communication constructs on multiprocessor systems to direct the dataflow paths. In this way, each SDF subgraph simply follows its own static schedule without concerning how dataflow is routed outside the subgraphs or changing its semantics and is therefore free from non-determinism.

So far, we have gone through the main line of dataflow computational models for stream processing including the semantics, scheduling and possible non-determinism. In the next section, several optimizations based on static analyses of process semantics, channel buffer management, and dynamic scheduling will be explored. Related work and their implementations will be introduced with only essential differences from SDF.

3 Optimizations

In dataflow models, stream programs are expressed as directed graphs in an appropriate granularity to explicitly uncover the flow dependencies and opportunities for parallelization. To justify the usefulness, compiled stream programs should run at least as fast as their sequential counterparts and consume comparable resources. However, a naive implementation of a stream compiler may slow down the stream program due to memory performance, scheduling and synchronization overhead. The memory performance overhead concerns cache locality as well as unnecessary copies when the dataflows are duplicated or reordered for downstream access without contributing to the computation directly. Scheduling and synchronization overhead are introduced when dynamic schedulers are used to determine the execution order and care must be taken to avoid races. On the other hand, the sequential counterparts may be written without involving such overhead at runtime and perform better. Therefore, optimizations for stream programs are essential. Moreover, while it is possible to extract the dataflow information from sequential programs, the dataflow nature of stream programs facilitates static program analysis for optimizations without considering the complexity of the sequential-to-dataflow transformations.

3.1 Overview

Given a stream program and its directed graph representation called a stream flow graph (SFG), optimizations for stream programs involve the manipulation of SFGs and scheduling, reductions of memory and arithmetic operations as well as dynamic workload adjustment. Table **??** lists existing stream processing optimizations and organizes them in terms of graph manipulation, semantics change, and application occasions (i.e., statically or at runtime). This should provide high-level ideas and

Optimization	Graph	Semantics	Dynamic
Operator reordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Load balancing	unchanged	unchanged	(depends)
Placement	unchanged	unchanged	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

Table 1: A catalog of stream processing optimizations in **Hirzel2011**

strategies for improving the performance of stream programs and facilitate new optimizations on uncovered areas. Particularly, some optimizations such as batching, algorithm selection, placement, load balancing and shedding are frequently referred to in later sections for discussions. In the following, a computing node in the SFG is termed operator, firing an operator is invoking its function and each optimization is briefly summarized to capture the notions.

Operator reordering assumes a pipeline of neighboring operators that includes an early operator that performs computationally intensive operations and a later operator that filter these results. Reordering the two operators to filter input items first will reduce the overall workload. *Redundancy elimination* assumes a branching of dataflow such that each downstream branch begins with the same operator. Pushing the common processing operator upstream eliminates redundant computations in each branch to save resources. *Operator separation* separates an operator with processing and selection logics combined into two operators in a pipeline in the order suggested by operator reordering to reduce the workload. *Fusion* assumes a pipeline of two operators with a lightweight latter one and merges the two into one single operator to avoid the overhead of invocations and data transfer in between. *Fission* increases data parallelism by pushing an operator to downstream branches and is the opposite of operator elimination. *Load balancing* is considered after fission to selectively dispatch incoming data items to idle or less busy replica branches to improve the overall throughput. *Placement* partitions the SFG by assigning operators to different CPU cores in proportion to their computational complexities. This optimization may be viewed as static load balancing and the CPU cores can be asymmetric on embedded platforms. *State sharing* assumes a common data source which is accessed in sliding windows of different sizes by multiple downstream operators and suggests to reuse instead of replicating the same source stream to save space and improve mem-

ory performance. *Batching* increases the channel capacities for large computationally intensive operators to execute more times to exploit instruction cache locality for better performance. *Algorithm selection* assumes an operator may be implemented using different algorithms good at different situations and replaces it when the condition changes. *Load shedding* filters out some less important input items sensibly to trade accuracy for performance because a flood of data streams may prevent the system from delivering timely responses. Apparently, some optimizations can be be combined together while others may conflict with one another. The applicability of stream optimizations also depends on the actual SFG, implying no single optimization guarantees the best performance.

In the remainder of the section, we will go through two stream processing toolkits that we have developed. CSense **Lai2013** is a stream-processing toolkit for mobile sensing applications (MSAs) based on dynamic scheduling. Next, we will present ESMS, a stream compiler that performs whole program analysis to optimize memory operations.

3.2 Dynamic Scheduling

Mobile phones are capable sensing platforms that include multi-modal sensors, increasing computational and memory resources, and versatile networking capabilities. Their capabilities have enabled a new generation of MSAs. We are interested in using mobile phones to transform how healthcare professionals collect information regarding a patients physiology, physical activities, and social interactions. Results of recent studies on mobile health systems have shown the feasibility of collecting medical records with higher resolution than is possible through manual data collection methods. However, experience has also shown us that the development of MSAs is particularly time demanding and challenging as significant time is spent on ensuring that the system operates robustly within the resource constraints of the platform.

The development of MSAs faces several challenges that are poorly addressed by operating systems such as Android:
Concurrency: MSAs must handle data processing and asynchronous events concurrently: sensors are sampled, data is uploaded to servers, and the system responds to user interactions or changes in the environment. Such systems are difficult to implement correctly using low- level concurrency primitives such as threads or events. Thus, MSAs require a flexible concurrency model that supports static analysis to detect bugs.
High Frame Rates: MSAs collect data from one or more sensors at high rates (e.g., 44100Hz for sound applications). The collected data frames must be processed in real-time or a few seconds from collection and may involve expensive signal processing operations (e.g., FFT). Supporting such high rates is difficult due to the limited

computational resources available on phones.
Reliability: MSAs are intended for long-term data collection from users in unpredictable environments. This operating regime, coupled with the need to provide a positive user experience, motivates a focus on bug prevention to reduce run-time errors.
Java Run-time Environment: Although Java increases programmer productivity and reduces programming errors (compared to C/C++), it also increases the complexity of implementing MSAs efficiently. Efficient implementations must manually manage memory, select appropriate concurrency mechanisms, and integrate native implementations of expensive operations.

In contrast with traditional SDFs, CSense adopts a dynamic scheduling approach to support MSAs for long-term energy efficient executions and process asynchronous events. As a stream processing toolkit targeting MSAs on the Android platform, CSense **Lai2013** prevents common programming errors w.r.t. memory allocations and concurrency control by providing a stream compiler that performs flow analysis at compile time to match the rates defined in SDF, partitions the SFG for asynchronous event processing, and generates runtime schedulers for exploiting target-specific power saving mechanisms. CSense provides high throughput with memory pooling and efficient thread synchronization.

A CSense application is represented as a SFG which may be partitioned into domains executed by domain schedulers. Each domain scheduler is executed on a different thread at runtime. A domain scheduler maintains a task queue to execute the SFG components in its domain in a depth-first traversal fashion and an event queue to process external events posted by other application components. Each component in the execution path follows the SDF semantics to execute only when all the inputs are ready, but allows to make a dynamic decision regarding the next component to schedule. For long-term power efficiency, the domain scheduler acquires the *wakelock* provided by the Android platform for applications to explicitly request CPU resources when determining there is sufficient data buffered to process and releases it otherwise. CSense allows programmers to annotate domain boundaries for partitioning without bothering thread synchronization and potential races. Instead of allocating channel buffers, CSense adopts message passing semantics to avoid unnecessary copying across channels. Though memory allocations are also taken care of by CSense, programmers are required to make sure each message flow path starting from a source component eventually reaches a tap for reuse in case of leaks.

Fig. **??** shows an example Speaker Identifier MSA which reads audio samples from the microphone, performs the RMS classifier to select only voiced samples to pass to the MFCC filter for feature extraction and then uploads the features to a remote server in the end. Since network access may block unexpectedly, the programmer annotates the httpPost component as a new domain such that CSense partitions the

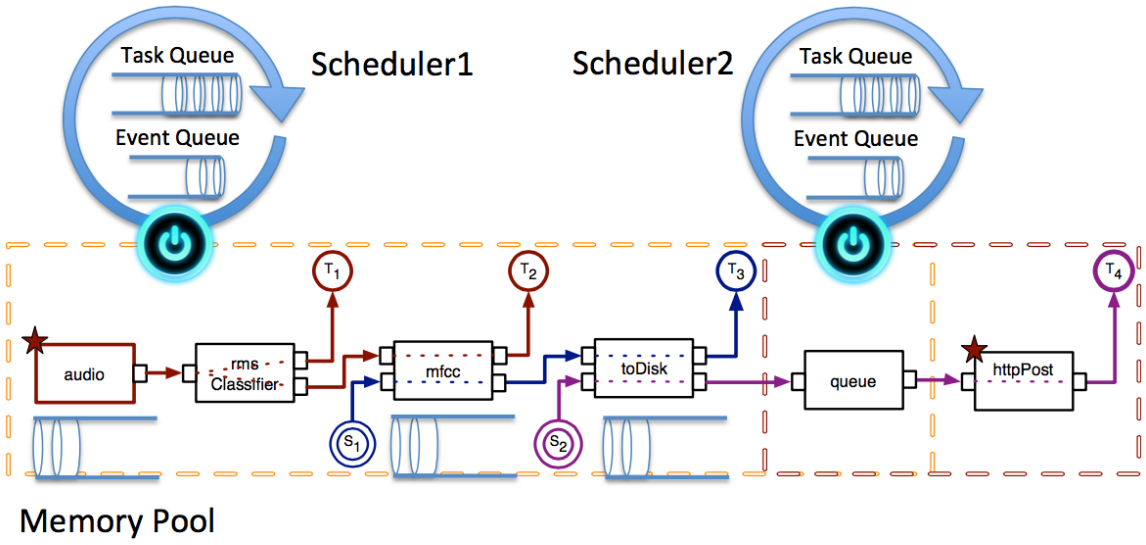


Figure 4: An example MSA: Speaker Identifier.

pipeline into two domains connected by a synchronous queue inserted automatically. The programmer also needs to identify all the possible memory flow paths associated with appropriate sources and taps. In this app, there are three memory flow paths, ⟨audio, rms, T1|mfcc, T2⟩, ⟨s1, mfcc, toDisk, T3⟩, and ⟨s2, queue, httpPost, T4⟩. The first component in each path must be a source component managing a message pool. Each time a source component is invoked, it takes a free message from the pool to use and pass down the path until the message reaches the tap which recycles the message by putting it back to the pool. The rms component in the first path is a dynamic component that may passes a message through either output ports. In this sense, the rms component implicitly partitions the SFG into three SDF subgraphs to implement asynchrony. The CSense compiler ensures the message in either path eventually reaches the tap. The performance evaluations in the following figures justify the design decisions lead to high performance for MSAs.

Fig. **??** shows the performance of a producer consumer benchmark with one producer component and one consumer component in a pipeline using different memory allocation and thread synchronization primitives. The x-axis is the production rate while the y-axis is the consumption rate. Ideally, both should agree but if there is a bottleneck the consumption rate drops. The experiment evaluates the performance overhead of memory allocations based on the Java runtime garbage collection (GC) and the CSense message pools (MP) as well as thread synchronization primitives based on the Java built-in re-entrant locks (L) and the CSense spinlocks (C). Surprisingly, the performance difference can be up to 19X, implying memory allocations

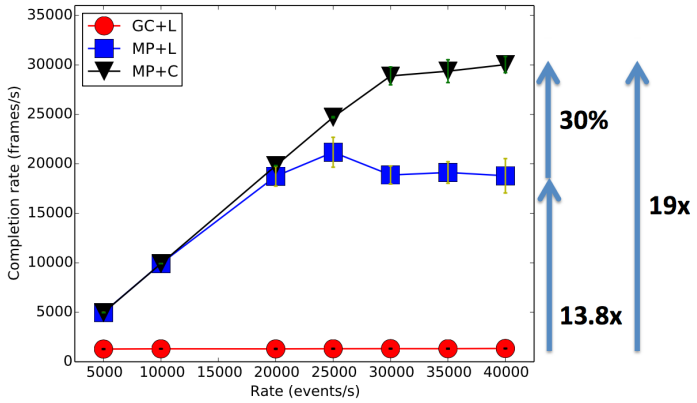


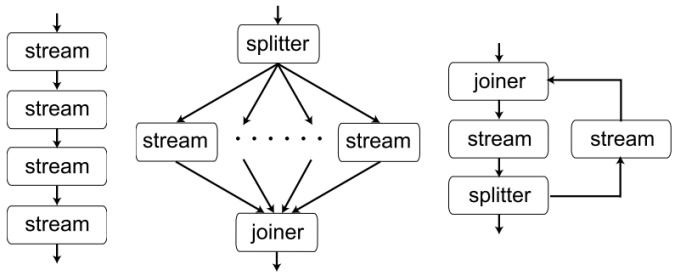
Figure 5: Performance improvement in terms of memory allocation and thread synchronization mechanisms.

based on GC on mobile platform are not desirable for high performance computing. Besides, multi-threading may not contribute to performance gains as expected if significant synchronization overhead is incurred between threads. From this perspective, a stream-processing toolkit such as CSense is quite essential to provides a reasonable programming abstraction that eases the composition of MSAs, ensures concurrency safety and offers high performance to run computationally intensive components while supporting target-specific power control.

3.3 Memory Optimizations

From the experience of CSense, memory performance seems to have a significant impact on the efficiency of stream programs. We decided to further investigate the memory performance of stream programs by developing our own compiler ESMS to analyze programs written in a well-defined stream language, StreamIt **Thies2002** StreamIt extends SDF and incorporates comprehensive static stream optimizations and benchmarks for comparison and evaluations. The StreamIt language defines streams as the first order construct in a SFG. A filter is a stream that is allowed to have at most one input and at most one output as well as implement a user-defined work function to process input. The work function declares the *peek*, *pop* and *push* rates, specifying the number of primitive stream operations performed on the input and output FIFO channels in one invocation. Those primitive stream operations directly correspond to $peek(i)$, $pop()$ and $push(v)$. $peek(i)$ reads a sample from the input channel at offset i without removing it while $pop()$ reads and removes the first sample from the input channel. $push(v)$ appends a sample v to the output channel. The three primitive stream operations are the only way to access input and output

channels in StreamIt programs.



(a) A pipeline. (b) A splitjoin. (c) A feedbackloop.

Figure 6: Three StreamIt hierarchical constructs.

To build a large stream program, StreamIt only defines three hierarchical constructs for simplicity, shown in Fig. **??**. Apparently, a pipeline represents a line of connected streams in order. A splitjoint begins with a splitter that either distributes input items according to a set of weights in a round-robin fashion or duplicates input items for each branch. Each branch of a splitjoin can be simple filters or other stream constructs. The joiner of a splitjoin must be round-robin by interleaving the input items according to the specified weights. A feedbackloop implementing a cycle begins and ends with a joiner and a splitter as in the splitjoin in a reverse order. For most MSAs, pipelines and splitjoins are sufficient to express the application semantics. We plan to support feedbackloops in near future.

StreamIt assumes a simple memory model with statically allocated buffer in each channel as SDF. However, we found potential inefficiency involving frequent rearrangement of data samples across splitjoins and some reordering filters. This makes sense because those structures never transform the data samples but moving them across channels tends to incur significant memory copy overhead and redundant allocations that may reduce cache locality. To address this issue, we assume a global memory allocation for all the channel buffers which should be allocated in contiguous pages by the OS and let each location in the layout represent an allocated live variable which can only be accessed through the aforementioned primitive stream operations. Each live variable is associated with a live range and occupies some location in the layout. The live range begins when a value is pushed to its location in the layout and ends when the value is popped from the location. A location in the layout is free if it is not occupied or the allocated variable is dead (i.e., outside its live range). In this way, the problem is formulated as determining the memory layout of a global allocation that minimizes the total memory requirements in terms of the allocation size and code complexity. Though a simple first-fit strategy suffices to find the minimum memory allocation, the resulting code complexity may be arbitrarily high due

to irregular access patterns across filter invocations and thus lead to increasing memory requirements. This is because the StreamIt memory model assumes a contiguous layout for each channel so that filters can execute in loops to access channels in the FIFO order. If the physical layout of the data samples are mapped to non-contiguous locations, the compiler needs to generate additional code to make sure of correct access in loops, increasing the code complexity accordingly. To search for an appropriate trade-off, we proposed three effective strategies to derive the minimal layout with reduced code complexity. The procedure follows.

First, our ESMS compiler interprets the memory operations for each filter in a given SFG in terms of their primitive stream operations in the work functions. If a value is popped from the input channel, it is interpreted as ending the live range of the corresponding live variable. If a value is peeked or popped from the input channel and then pushed to the output channel without being modified, the memory operation is interpreted as *pass* introducing no copy overhead. Otherwise, it is an *update*, writing a new value to the output channel at a specified location. Each interpreted memory operation is associated with a logical location in the channel.

Next, to relate the logical locations to physical locations in the global layout, a complete period of the static schedule is simulated by applying the interpreted memory operations to the global layout. Whenever a value is pushed to the layout, the ESMS compiler checks if the $push(v)$ is a *pass* or *update*. If it is a *pass*, this is viewed as a live variable holding the value and occupying some location in the layout resumes its live range. Otherwise, it is an *update* which can be viewed as a live variable being created and starting its live range at some free location in the layout. If the live variable at the specified location just ends its live range, the location may be reused without growing the global allocation size. Or else the global allocation size may be expanded to provide a free location for the new live variable. The ESMS compiler employs three heuristics to deal with the latter situation, Append-Always (AA), Append-on-Conflict (AoC) and Insert-in-Place (IP). The AA strategy always grows the layout size for the new live variable to occupy. The AoC strategy grows the layout to store all the new live variables created by the *updates* in the current invocation window only if any of the *updates* cannot reuse a free location. The IP strategy reuses free locations whenever possible and otherwise inserts the new live variable in place instead of appending at the end of the layout in expectation of better locality for downstream access. In this way, the ESMS compiler captures the live ranges at each physical location in the global layout and determines the allocation size in the end.

Subsequently, we compare the performance of ESMS using the three strategies with the StreamIt compiler w/ and w/o *cacheopt* enabled on the Intel x64 platform in terms of code and data segment sizes allocated in 4K pages reported by the command line utility *size* as well as speedup in the selected benchmarks. StreamIt w/o cacheopt

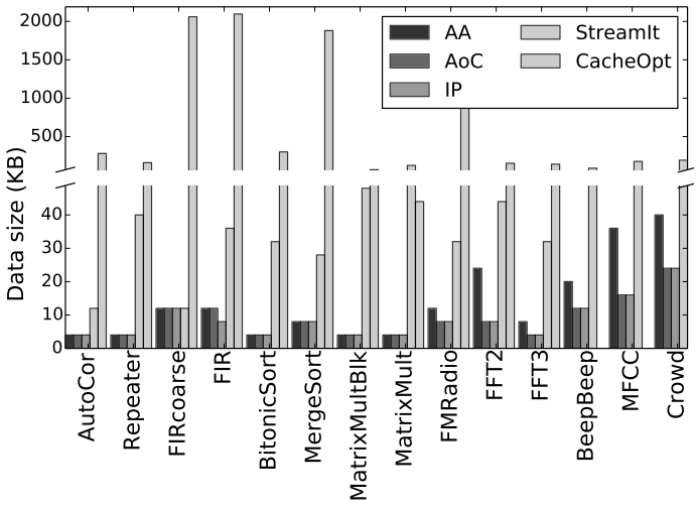


Figure 7: Data Segment Sizes.

serves as the baseline. StreamIt w/ cacheopt (CacheOpt in short) enables its cache optimization which implements the batching technique in section 3.1 to allow filters to execute more times in expectation of exploiting cache locality at the expense of larger channel buffer allocations.

Fig. **??** shows the data segment sizes for each compiled benchmarks. As expected, AA causes larger data segment allocation and AoC is comparable to IP which potentially allocates the least. The average reduction in data segment size of AA, AoC, and IP are 50KB, 55KB and 98KB which represent 4596% reduction in percentage compared to the baseline and CacheOpt. The results imply the splitjoins in StreamIt programs are extensively used to route dataflows and contribute to increasing channel buffer allocations. On the other hand, the ESMS compiler captures the physical locations mapped by primitive stream operations so that the channel buffers associated with the splitjoins and even reordering filters can be completely eliminated to save space.

Fig. **??** illustrates code segment size reduction. The average code reductions for AA, AoC and IP are 130KB, 136KB and 143KB. In relative terms, the average reductions are 69%, 72% and 77% respectively. The ESMS optimization reduces the code size by eliminating splitjoins and reordering filters since all their semantics are captured and remapped for downstream access. Nevertheless, even with these savings, there are cases when some of the ESMS strategies might generate larger code size than StreamIt due to increased code complexity to deal with irregular access patterns, though the total memory requirements with code and data segments combined are still reduced. On the other hand, CacheOpt typically has a minimal impact on the

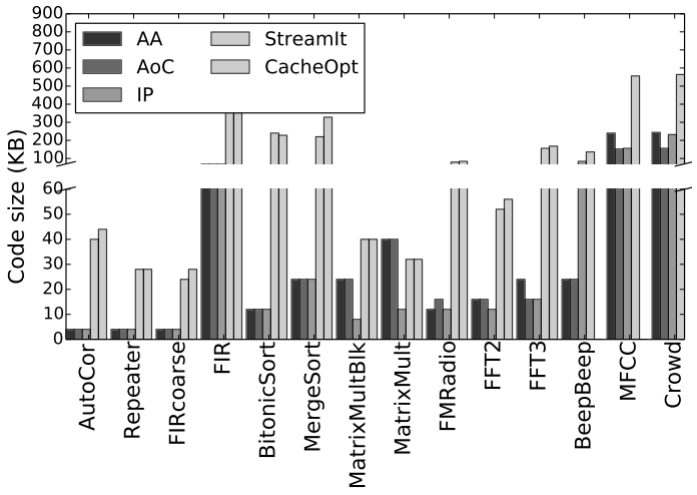


Figure 8: Code Segment Sizes.

code size. On average, it adds 14KB to the code size in the selected benchmarks.

Fig. **??** shows the speedup relative to the baseline StreamIt. The average speedup of AA, AoC, and IP are 3, 3.1, and 3. In contrast, the average speedup of CacheOpt is merely 1.07. All of ESMS heuristics outperformed CacheOpt with the exception of the two FIR benchmarks. The reason for these significant processing time improvement is the fact that ESMS uses less memory access by effectively sharing data across components. We validated that this was the case by using *cachegrind* to track the number of memory references. Moreover, the smaller memory footprint leads to a smaller working set that fits within the L1 cache, implying fewer misses. On the Intel platform, the ESMS compiler managed to achieve significant improvement in terms of both reducing memory consumption and increasing stream processing throughput. AoC and IP that handle live range conflicts either by appending or inserting achieve better resource usage than AA across all dimensions. Resolving conflicts through insertion tends to achieve more reductions in data but slightly lower speedup than appending.

To sum up, in contrast with the StreamIt cache-aware optimizations in **Sermulins2005** that trades space for performance, our approach saves space while eliminating unnecessary memory operations to improve the performance. In **Bhattacharyya1994** the memory reuse is based on overlaying channel buffers in terms of their live ranges while maintaining periodical modulo access for ring buffer usage. Compiler optimizations were also considered to generate instructions to avoid the modulo overhead selectively given the static schedule. From this perspective, our ESMS compiler optimizations allow for more aggressively reuse and even remap non-contiguous memory accesses across filter invocations at some cost of increasing code complexity. Other mem-

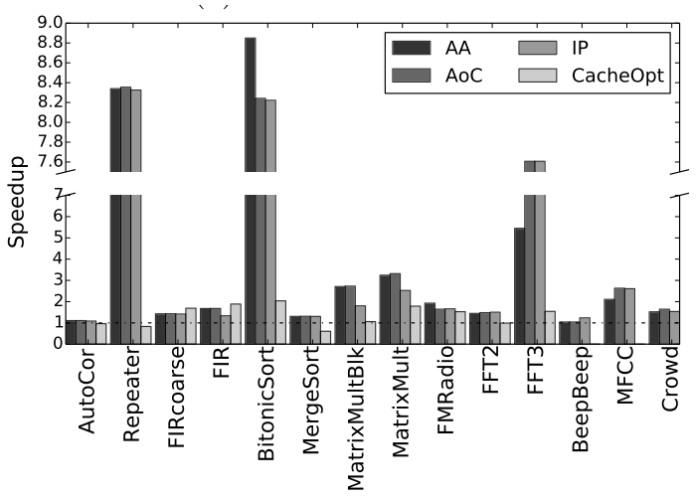


Figure 9: Speedup.

ory management techniques for stream processing include XStream **Girod2008** and StreamFlex **Spring2007** The former proposed *SigSegs* to merge and segment data stream items with zero copy overhead and practice copy on write at runtime. This technique based on two-level linked lists requires runtime memory management which is likely to burden GC and should be avoided by MSAs, not to mention its dynamic scheduling incurs switching overhead between stream operators. On the contrary, our ESMS compiler resolves live range conflicts at compile time and avoids copy overhead introduced by splitjoin and reordering semantics. StreamFlex is yet another stream processing toolkit written in Java, aiming at avoiding the GC overhead while satisfying real-time constraints. Its memory model relies on a customized region-based allocation which allocates objects in a special memory area and is free from the default Java garbage collection. However, it is generally impossible to customize such a memory model on popular Android devices. Later, we are going to introduce the linear analysis in **Lamb2003** which is an effective alternative to improve the performance by reducing the number of linear filters while saving memory usage accordingly.

3.4 Linear State Space Analysis

In the signal processing domain, the output of a linear filter can be derived from the linear combination of its inputs. Given such linear filters represented as matrix multiplication forms, neighboring linear filters can be combined by multiplying both matrix forms. This improves performance by reducing the number of arithmetic operations and data transfers between filters. Fig. **??**. illustrates an example StreamIt filter and the extracted linear form $\lambda = \{A, \vec{b}, e, o, u\}$ where A is

the matrix recording the coefficients for linear combination with the inputs, $\vec{b}$ is a constant row vector to add to the linear combination and e, o, u represent peek, pop and push rates of the filter. The rows of A is in the peek order from bottom to top while the columns of A is in the push order from right to left. Thus, in this example, the input vector $\vec{x} = [peek(2), peek(1), peek(0)]$ and the output vector $\vec{y} = [push(pos = 1), push(pos = 0)]$ are in the same peek and push order of A respectively.

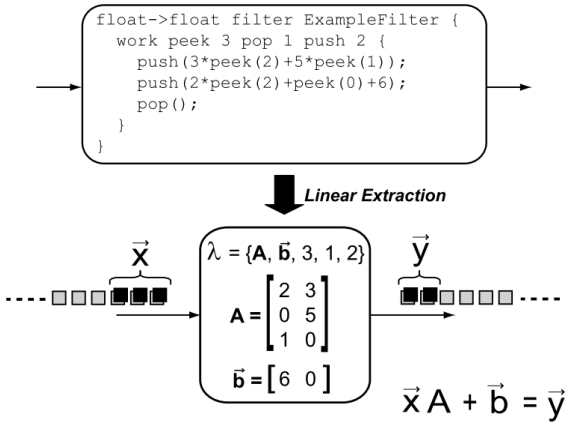


Figure 10: A linear filter and its linear form.

In this way, it is not hard to imagine how neighboring linear filters in a pipeline are combined. However, if two neighboring filters do not have consistent push and pop rates in between, linear expansions of either or both filters are necessary to linearly combine the matrix forms. The linear expansion works by duplicating the matrix A in the diagonal of an expanded matrix as well as changing the peek, pop, and push rates appropriately. Fig. **??** shows the steps to combine linear filters in a pipeline through linear expansion. The left filter needs to execute three times to generate enough output items for the right filter to peek. Therefore, the linear expansion is employed to the left filter and its coefficient matrix A is expanded along the diagonal while the peek and push rates increase to match the rates of the right filter. The last step of pipeline combination is straightforward by multiplying both possibly expanded coefficient matrices A_1^e and A_2^e to derive A'.

Similarly, splitjoins can be linearly combined if all the branches are linear streams. Consider a duplicate splitjoin with linear filters in both branches as in Fig. **??**. The right branch is expanded twice to match the push rate of the left branch and the round-robin weights of the joiner. Since the joiner interleaves the outputs from both filter branches, the linearly combined matrix A' is derived by interleaving two columns of A_1^e and one column of A_2^e in the order from right to left twice. If the splitter is round-

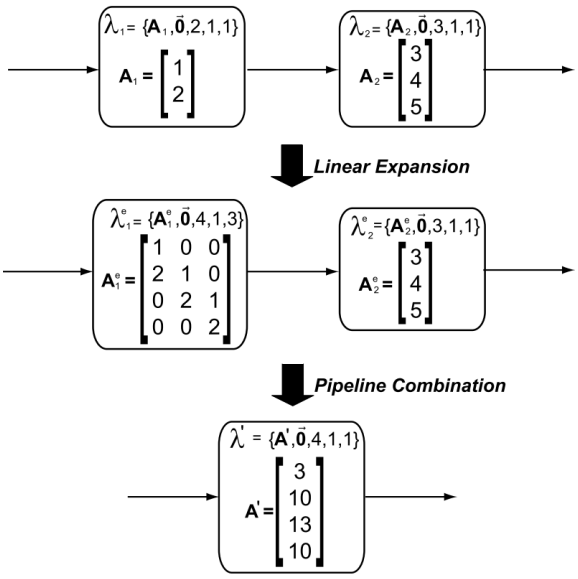


Figure 11: Combining linear filters in a pipeline.

robin, it is transformed to a duplicate one and linear decimeters are inserted to each branches to act as a mask to select the output samples such that the resulting output is the same as before the transformation. In this fashion, the linear combination of duplicate splitjoins can be applied to derive the linear combination of round-robin splitjoins.

Further optimizations take frequency domain transformation and linear state spaces into account. The frequency domain transformation of linear filters allows for linear combinations to be done in the frequency domain. The StreamIt compiler weighs the cost of whether performing linear combinations in the time domain or frequency domain by counting the number of required arithmetic operations to decide the optimal configuration of linear transformations. On other hand, the linear state space deals with stateful linear filters. The idea is to view states as stored in self-feedback loops such that the output can be derived from the sum of both linear combinations of input and current states. Likewise, states are updated as the output from the sum of both linear combinations of input and current states. Some of the states may be merged to reduce the number of total states and thus improve memory performance while saving resources. Clearly, our ESMS optimization is able to complement the linear state space analysis even for non-linear filters. This suggests the stream compiler has to take care of the dependencies between optimizations and each optimization should specify its requirements on the input SFG, channel buffer layouts and so on. In the next section, we will explore energy saving techniques rel-

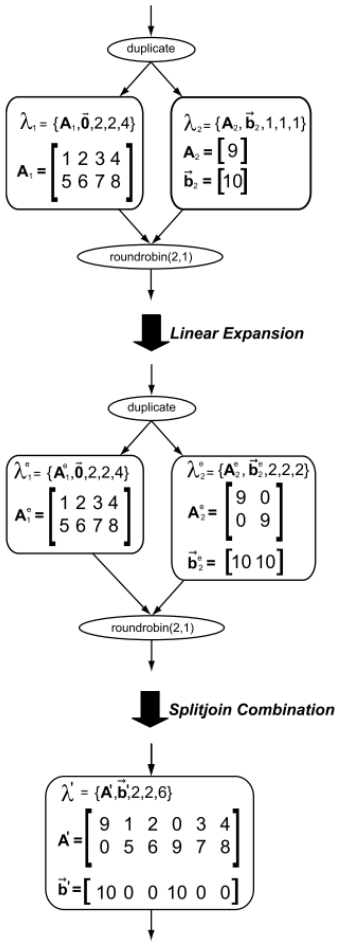


Figure 12: Combining linear filters in a splitjoin.

evant to stream processing in the domain of MSAs. In some cases, compared with sequential counterparts, stream programs could formulate the power requirements more precisely, leading to energy efficiency without sacrificing performance.

4 Energy Efficiency

In the MSA domain, energy consumption is obviously an important topic for continuous sensing because of the slow growth of battery life for years. It is therefore essential to understand the root causes of energy consumption in MSAs on mobile devices from the perspective of software behaviors. Next, since the design and implementations of most MSA are based on some stream processing models, it is reasonable to study the

impacts of memory, parallelism, concurrency and synchronization on energy consumption. Subsequently, we will explore several practical methodologies for energy savings ranging from dynamic voltage frequency scaling (DVFS), approximation programming, dynamic control, and code offloading. How those approaches benefit stream processing are evaluated in the end.

To understand the root causes of energy consumption on mobile devices, an effective profiler such as the *eprof* **Pathak2012** is worth studying. Their experiments show that up to 75% of energy consumption may be attributed to third-party advertisement modules in free apps. Otherwise, improper usage of the Android power management primitives tends to kill the battery life unexpectedly. Android provides the wakelock for apps to explicitly acquire CPU resources. However, its usage mimics the condition variables and is similarly error-prone, causing races when used by multiple threads. Forgetting to or late releasing the wakelock inevitably wastes more energy than necessary. It is essential that programming toolkits such as CSense **Lai2013** automatically manage power components. Alternatively, Hu, Pathak, Abhinav, et al. **Hu2012** proposed to analyze the code paths and identify unreleased power components though programmers still need to fix the code manually.

I/O Components. The remaining causes of energy consumption is due to the I/O components other than CPU or memory including GPS, SD card, WiFi NIC, cellular, bluetooth, sensors and speakers. An I/O component usually has several power states and exhibits asynchronous power behaviors. When the I/O component is serving a request, it enters a high power state and consumes more energy. However, after the request is done, the I/O component does not shut down immediately but enters a tail power state and continues consuming energy for a while. This gives the idea similar to batching that I/O component requests should be aggregated and scheduled to perform within a short interval so as to avoid tail energy waste as much as possible. In stream processing, I/O request batching can be applied to those components interacting with I/O devices. Nonetheless, external negotiation with other apps in the system would be necessary for overall energy savings.

Synchronization. While a main selling point of the dataflow model is exposing explicit parallelism of programs, the impacts of parallelism, concurrency and synchronization on energy consumption are not clear yet. On one hand, high performance stream processing shortens the execution time and thus should save energy consumption on the CPU. On the other hand, multi-cores activated simultaneously by parallel programs may consume more energy. A pattern-oriented view in **Liu2012** provides useful insights in the granularity of threads that may guide further stream processing optimizations. In essence, energy is wasted in parallel and concurrent programs when processes or threads are not contributing to the throughput but waiting for synchronization. To be specific, the energy is wasted when processes or threads are waiting, either spinning or blocking. Spinning spends CPU cycles without producing results

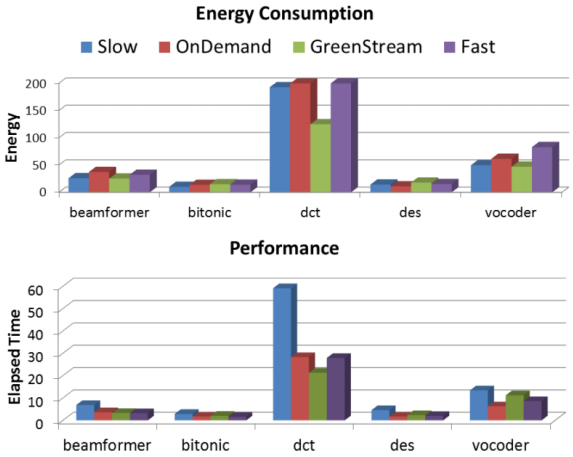


Figure 13: Energy savings with DVFS on stream programs.

while blocking induces context-switch which potentially leads to cache misses and consumes more energy. What causes the waiting is mainly the relative speed differences between processes and threads in their execution paths with different workload. However, not all the waiting can be avoided. Sometimes, a thread is simply created to wait for other worker threads to complete. An intuitive idea is to dispatch the waiting threads to a core that is configured to run slower. In control flow languages, programmers exploit parallelism with threads following commonly used design patterns. By identifying the patterns used, those waiting threads may be inferred. However, there might be innumerable patterns for all kinds of situations and it might not be trivial to recognize the patterns in a general way. On the contrary, the synchronization points in dataflow models may be located around the partition boundaries. Previously mentioned placement and load balancing optimizations aim to balance the workload and consequently improve the throughput by reducing the process waiting. Nevertheless, placement is static and load balancing focuses on data parallelism. It would be preferable to cover pipeline parallelism and adapts to changing workload given the fixed placement.

DVFS. In stream processing, if components produce very different throughput, some critical path may exist as the performance bottleneck such that other non-critical path components waste energy for the CPU running at a higher frequency than necessary. A promising solution to throughput differences in stream processing is using DVFS. A recent experiment based StreamIt in **Bartenstein2013** shows energy savings up to 28% are achieved with DVFS without sacrificing performance by estimating the workload in terms of profiled natural rates. Recall a StreamIt filter declares its peek, pop, push rates in one invocation. The natural rates of the filter is

its throughput of consuming input and producing output respectively per unit real time. From the perspective of pipeline parallelism, energy efficiency is reached when the natural rates of neighboring filters agree. The authors profile the natural rates at the highest CPU core frequency and assume the rates decrease in proportion to the frequencies. The CPU cores only provide a limited number of scaling steps. So the authors formulated a linear program to capture the constraints and determine the core execution frequencies for the placement such that some of the cores do not need to run at the highest frequency to maintain the overall throughput. Fig. **??** illustrates the energy consumption and performance across the selected benchmarks. *GreenStream* represents the energy efficient configuration while *Fast* means all the cores run at full speed and *Slow* at the lowest. *OnDemand* is the default Linux power management policy. The results clearly justifies the claim. Some other concerns include the thread start-up time but it might not be the case with stream programs which assume the external input comes continuously at least for a sufficiently long duration and the placement partitions the SFG properly. In this sense, stream programs are able to request CPU resources with more precise information w.r.t. the workload such that mobile platforms practicing aggressive power control are likely to benefit from appropriately dispatching CPU cores to run stream programs at lower frequencies without sacrificing user experience in terms of performance.

	Mild	Medium	Aggressive
DRAM refresh: per-second bit flip probability	10^{-9}	10^{-5}	10^{-3}
Memory power saved	17%	22%	24%
SRAM read upset probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
SRAM write failure probability	$10^{-5.59}$	$10^{-4.94}$	10^{-3}
Supply power saved	70%	80%	90%*
`float` mantissa bits	16	8	4
`double` mantissa bits	32	16	8
Energy saved per operation	32%	78%	85%*
Arithmetic timing error probability	10^{-6}	10^{-4}	10^{-2}
Energy saved per operation	12%*	22%	30%

Table 2: Hardware approximation strategies and savings.

Memory. Popular mobile platforms such as the Android provide the JVM runtime to host applications. The implications of energy consumption imposed by the memory behaviors of the VM such as byte code interpretation, class loading, garbage collection (GC) and dynamic compilation are investigated in **Vijaykrishnan2001** In summary, from the perspective of cache configurations, larger cache size and associativity tends to reduce energy consumption for fewer memory accesses. However, as long as the cache is able to contain the entire working set, larger cache size instead incurs additional energy consumption per cache access. In the software aspect, mem-

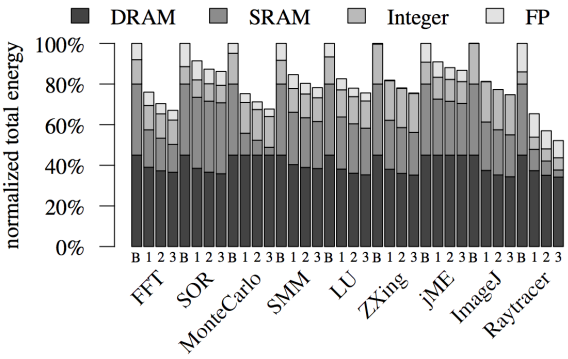


Figure 14: Benchmark energy consumption for each strategy where B is baseline w/o approximation and 1 to 3 represent the three strategies.

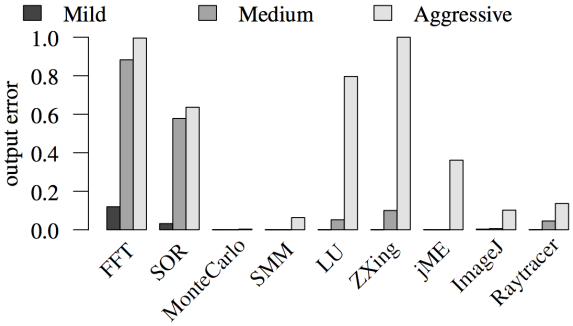


Figure 15: Benchmark output error for each strategy.

ory access instructions including load and store constitute about 20% of the total instructions in the benchmark suit but account for more than 50% of the total energy consumption. Clearly, an energy efficient program should maintain a smaller working set while reducing the number of memory references as possible. In this way, our ESMS memory optimizations proceed on the right track.

Programming. Recently, a novel idea about energy savings is the approximate computation which trades accuracy for energy savings. A similar stream processing optimization would be the load shedding that discards some input when busy at the expense of accuracy. In contrast, approximate computation does not filter out the input but computes the results and stores the data in a less accurate way with low-power storage, low-precision operations and energy efficient algorithms. EnerJ **Ceze2011** is a language extension providing annotated types to allow programmers practicing approximate computation only when necessary and safe. This simplifies the analysis to isolate parts of the program that require to be precise and those that can be approximated. Table **??** lists hardware approximate strategies used in storage

and operations along with the savings in percentage. Fig. **??** and Fig. **??** demonstrate the resulting energy consumption and output error in the selected benchmarks respectively. Though languages such as EnerJ provides fine-grained approximation, it may be simpler to annotate in the granularity of filters in stream programs such that all the downstream components apply the same approximation. When combined with the optimization of algorithm selection, runtime switch between levels of approximation in view of system load can be useful and flexible.

Dynamic Control. Other energy saving methodologies typically involve dynamic control which weigh relevant trade-offs at runtime to make energy efficient decisions of sampling, batching and code offloading. Symphoney **Ju2012** proposed to respect foreground applications without sacrificing user experience by adjusting sampling rates and periods in the units of high level application frames. This makes sense considering from the perspective of contexts instead of low level samples when the system is overwhelmed. Apparently, Symphoney shows the way to realize load shedding with graceful degradation. While batching is an effective stream processing optimization, it also saves energy consumption by reducing the duration of tail power state despite larger memory usage. However, the quality of service (QoS) required by applications are of the same importance. APE proposed in **Nikzad2014** allows programmers to defer the processing of power hungry code segments until the specified desirable conditions are met while respecting the deadlines. In stream programs, it would be easier to identify power hungry dataflow paths through whole program analysis and free programmers from manual annotations.

Code Offloading. As for code offloading, earlier on, mobile devices are not capable enough and heavy processing is offloaded to remote cloud computing infrastructures. This technique essentially trades the communication power for energy savings of local computation. MAUI **Cuervo2010** demonstrated how to weigh the cost by formulating an integer linear program and achieved fine-grained energy-aware offload of mobile code. Recently, alternative code offloading options including asymmetric multi-processing (AMP) CPU and GPGPU cores are available. In this case, a stream compiler may opt for partitioning stream programs to run on local heterogeneous cores for more energy savings but dynamic compilation techniques might be incorporated for architecture-specific optimizations.

5 Cloud Stream Processing

Stream processing achieves high throughput and low latency by applying graph transformations and static analyses on memory management as well as scheduling in a managed runtime environment where consistency and fault-tolerance are typically less concerned. In contrast, cloud computing requires to process incoming data streams at

extremely high velocity in a distributed setting where consistency and fault-tolerance are essential trade-offs with performance. Further, one large class of algorithms to run on cloud infrastructures includes machine learning and data mining which expose iterative and convergence graph-parallel computation characteristics rarely seen in traditional stream processing contexts. In this section, we introduce two dataflow systems, Kineograph **Cheng2012** and Naiad **Murray2013** The former allows for continuously changing graph structures while maintaining consistency through global progress tracking and fault-tolerance efficiently. The latter realizes timely dataflow applications by providing synchronous and asynchronous programming constructs leading to high throughput and low latency performance. Finally, we consider the performance of distributed graph-parallel computation when switching between synchronous and asynchronous execution modes appropriately.

5.1 Kineograph

To address the requirements of dealing with continuous input streams, updates as well as changes to graph structures and producing timely insights into fast incoming data sets, Kineograph **Cheng2012** decouples graph computation from graph structure updates and employs an epoch commit protocol which takes consistent snapshots progressively to ensure the global progress towards convergence. Other techniques such as quorum-based replication, re-execution to recover node failures and primary-backup of final computation results are also incorporated but only the epoch commit protocol is detailed in this section.

The entire system of Kineograph is composed of ingest nodes and graph nodes. The ingest nodes accept and analyze external raw data feed in records, creating transactions of associated graph update operations to distribute with sequence numbers to graph nodes. The graph nodes store the graph updates but postpone until the arrival of a snapshot that includes the transactions to apply. In the meanwhile, the ingest nodes also report the graph update progress in a global progress table containing a vector of sequence numbers as a logical clock timestamp maintained by a central service so as to allow a snapshooter to periodically instruct all graph nodes to take a snapshot of the current logical clock value. The logical clock timestamp of a snapshot defines the end of an epoch. To commit the epoch, graph nodes perform all the stored transactions of local graph updates in order up to the epoch and produce a graph-structure snapshot for further graph computations. In this way, graph updates are decoupled from graph computation based on static graph-structure snapshots in an epoch-granularity.

Next, to facilitate graph computations, graph nodes is divided into a storage layer and a computation layer. In the storage layer, the graph nodes are partitioned logically and assigned to physical machines by hashing their vertex ids without locality

consideration. The storage layer makes sure to provide consistent snapshots of graph structures across partitions for graph computations. In the graph computation layer, Kineograph adopts the vertex-based model and user-defined rules to check if there are changes of local topology of vertices compared with the previous snapshot and state updates from other vertices. If that is the case, a set of user-specified functions associated with the vertex is reevaluated. If the results significantly differ, the new values are propagated to a user-defined set of vertices and optionally go through graph-scale aggregation for computing global properties. The propagation of state updates and reevaluation of vertices terminate when no more changes are identified. Fig. **??** visualizes the computation flow. To better support various graph-mining algorithms, Kineograph provides both the push and the pull models for propagation of vertex states. The push model allows vertices to actively send incremental state updates to relevant recipients which can be aggregated to reduce unnecessary communications. In the pull model, vertices read states of neighbors to update its values and notify the neighbors if the change is significant. The runtime scheduler maintains a task queue and schedules vertices across partitions requiring reevaluations due to local or neighbor state updates. In summary, vertices never directly change each other's states but post state changes for relevant neighbors to be scheduled for reevaluations and thus write races are prevented.

The insights of Kineograph is the layered design that decouples fast changing states and relations from stable infrastructure services. In this sense, conventional stream processing and optimizations opting for static dataflow paths should efficiently serve the infrastructure services for the fast changing information flow to propagate. This implicitly aggregates communications and prevents media contentions between nodes. However, the dynamic information flow may differentiate workload across nodes such that some nodes become bottlenecks over time. In this case, in additional to load balancing and shedding, dynamically creating dataflow paths would be necessary and also account for node failures.

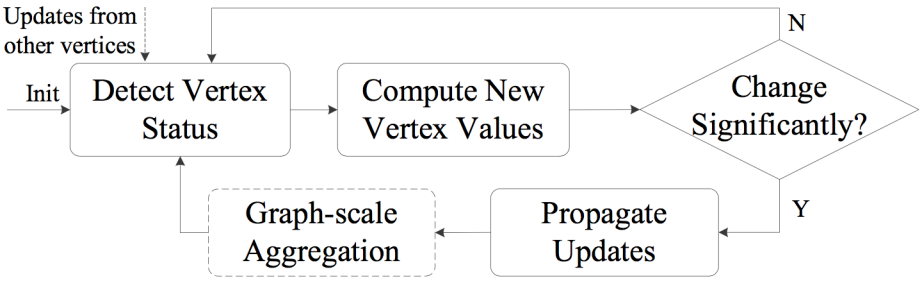


Figure 16: Kineograph computation flow.

5.2 Naida

Combining synchronous batching and asynchronous streaming, Naiad **Murray2013** allows to program dataflow systems with high throughput and low latency for iterative and incremental computations. Unlike Kineograph that maintains the graph structure over time, the dataflow graph in Naiad models the processing stages as nodes linked through connectors logically. The realization of a dataflow graph may map a processing stage to a set of vertices assigned to run on different machines for parallelism while the connecter between stages may route messages according to a user-defined partitioning function. In this sense, Naiad develops stream programs in a distributed way by providing fine-grained low latency synchronous notifications and asynchronous message delivery. For asynchronous message delivery, v.SENDBY(*edge, msg, time*) allows a vertex v to send a message along an outgoing *edge* with a timestamp *time*. The runtime schedule delivers the message to vertex u by calling u.ONRECV(edge, msg, time) at an appropriate time according to some scheduling policy. To provide synchronous notifications, a vertex v allows to request one by calling v.NOTIFYAT(*time*) and receive the notification when no more messages with timestamps less than or equal to time arrive and v.ONNOTIFY(*time*) is invoked. To ensure low latency and correctness, Naiad must track the progress of message delivery of a particular timestamp and determine the right time to deliver the notification. Since the dataflow graph allows to have nested loop structures, Naiad has to track the relations between messages even in loops. To achieve the goal, Naiad proposed a structured timestamp to label messages in the format $(e, \langle c_1, c_2, ..., c_k \rangle) = (epoch, loopvectorclock)$ where *epoch* is a monotonically increasing integer assigned by external data producers and the loop vector clock represents the number of loop iterations this message has gone through for each nested loop in order. The number of loop vector clock counters is expanded before entering a loop and removed after leaving the loop. The corresponding loop counter in the loop vector clock is incremented during iterations. In this way, the timestamp captures the partial order in time and space of related messages in the same epoch, denoting the progress of message delivery. Naiad proves that there exists a message frontier indicating no precursor messages in the same epoch along the message flow path. Therefore, it is straightforward to deliver the notification only when the message frontier reaches the vertex requesting this notification by tracking the message frontier in the same epoch w.r.t. the notification timestamp. To approximate the progress of the message frontier for vertices on different machines and processes, each time a message delivery event occurs, the information is broadcast to neighboring machines and processes to update their frontier approximation. The implementation of Naiad tweaks the default TCP/IP timeouts and narrows down the broadcast range to achieve low latency despite packet loss. Other optimizations involve the avoidance of garbage collection with static allocations and spinlocks for

low latency contention. However, the event-based programming model may not be productive for programmers to develop large dataflow programs quickly and easily. Some frameworks on top of the timely dataflow such as GraphLINQ are designed to facilitate the development of distributed dataflow applications on top of Naiad.

The insight given by Naiad is the timestamp like structures that form a partial order for some progress tracking to propagate only locally. Besides differentiating input streams in different epochs, it would be useful to locate nodes relevant to a particular computation for reconfiguration and termination in the dataflow order. For example, multiple Naiad applications may coincidentally employ the same computation and each of them terminates asynchronously such that some machines have computational resources allocated but no longer used. In this case, it would be efficient to propagate the application termination messages along the dataflow path in its partial order without interrupting any ongoing processing and a machine should release the resources only for the termination messages from the last leaving application. Alternatively, the conventional algorithm selection optimization at the scale of cloud computing involving multiple nodes may be applicable through some reconfiguration messages in some partial order.

5.3 Execution Modes

Large-scale graph-structured algorithms such as PageRank **Brin1998** allow to run either synchronously or asynchronously. In this case, Naiad combining both modes for execution across processing stages may not achieve the best performance. Xie et. al. **Xie2013** characterized the performance variations between both execution modes in relation to relevant contributing factors and proposed a mechanism to automatically switch at runtime for better performance. For iterative and incremental graph-parallel computations, synchronous execution (Sync) means updates are propagated to other vertices at the end of each iteration in a batch while asynchronous execution (Async) implies no explicit synchronization points and updates are visible to neighboring vertices as soon as possible. Table **??**. contrasts the properties of both execution modes. Apparently, Async has irregular communications across vertices and usually converges faster but it may not be the case if too frequent communications cause contention of network resources. In this sense, Async favors CPU-intensive algorithms for less message exchange as well as low workload for fewer active vertices, and scales up easily with the size of cluster. On the contrary, Sync performs I/O-intensive computations better with message batching under higher workload with a larger number of active vertices.

In view of the respective favorite scenarios for Sync and Async, execution under one single mode is unlikely to achieve the optimal performance given an unfavored scenario considering different algorithms, graph size and machine scales. Moreover,

	Sync	*Async*
Properties		
Communication	Regular	Irregular
Convergence	Slow	Fast
Favorites		
Algorithm	I/O-intensive	CPU-intensive
Execution Stage	High Workload	Low Workload
Scalability	Graph Size	Cluster Size

Table 3: Comparison between Sync and Async modes

even for the same algorithm, Sync and Async may show distinct performance in different execution stages. For instance, the single source shortest path (SSSP) has fewer active vertices in the beginning and end when Async performs better but more active vertices in the middle execution stage when Sync delivers higher throughput. On the other hand, some algorithms only converge in one execution mode such as the graph coloring which only converges under Async mode because Sync fails to break the symmetry. Nonetheless, Sync still accelerates convergence before the final stage. This suggests dynamic switching between the execution modes is essential for the best performance.

To support hybrid-synchronous execution, the graph-parallel processing architecture should incorporate both Sync and Async execution engines. The switching mechanism must ensure the consistency of graph structures, scheduling queues and pending update messages after switching. For efficiency, the data structures used in both modes should share as much as possible or at least the conversion incurs little overhead. For high performance, the switching timing needs to be predicted accurately to benefit from throughput gains and too frequent switches should be avoided because of the overhead of mode switching. To make a decision in terms of performance improvement, it is essential to estimate the throughput in the number of active vertices processed in unit time under the current mode and predict the performance under the alternate mode. If the current mode is Sync, the throughput is estimated as the number of active vertices in the next iteration divided by the iteration computation time plus the barrier synchronization time. Since the barrier synchronization time per vertex is nearly constant, the computation time per vertex may be predicted from the previous iteration with an appropriate learning factor. If the current mode is Async, simply remove the barrier synchronization time and estimate the computation time per vertex as in Sync mode. To predict the throughput under the alternate mode,

online sampling is adopted for higher accuracy. If the alternate mode is Async, simply switch and measure the throughput in a short time interval. If it is Sync mode, sample the increment speed of active vertices in the scheduling queue and compare it with the current Async throughput. When the number exceeds the current Async throughput, it implies the system is overloaded and it is time to switch to the Sync mode. Other heuristics such as specifying the threshold of performance gains to switch and the convergence mode for the final stage also apply when necessary.

In this way, programmers are free from the difficulty of deciding which execution modes to run the algorithm for achieving better performance and coding the execution mode explicitly. This decision making implies pure static analysis may not suffice in the context of cloud stream processing due to consistency, synchronization and communication trade-offs as well as execution stage variations over time. However, we believe with appropriate partitioning and layering, static analysis and conventional stream processing optimizations would still be applicable and complement the runtime decision making for overall improvement.

6 Conclusions and Future Work

Stream programs based on dataflow models allows to identify flow dependencies and captures parallelism explicitly. Each node in the SFG implements a sequential program in some imperial language. It is suggested to have filters of fine-grained functions such as adders, subtractors, multipliers and so forth since the stream compiler is able to fuse neighboring filters whenever possible. In this way, filters requiring developers with domain-specific knowledge such as FFT can be composed of fundamental filters written by ordinary programmers. Nonetheless, it might not be trivial for stream compilers to decompose large-grained filters in the level of semantics such as separating linear logics from non-linear parts and thus potential optimization opportunities may be lost. This raises a research question. Given two equivalent stream programs at different granularity, is it possible to efficiently derive the program transformation from the coarse-grained one to the fine-grained one? Examples include the FFT benchmark in two granularities in Fig. **??** and Fig. **??**. If it is ready, a coarse grained sequential version suffices for all kinds of stream optimizations. A following question is how much parallelism can be extracted from a given program. In other words, whether or not there is a dataflow representation of a program that exposes every possible parallelism. We believe the ultimate dataflow representation of a program would approximate the theoretical speedup given by Amdahl's law.

On the other hand, parallelism in practice does not always imply performance improvement but plenty of trade-offs and constraints to weigh and satisfy such as energy efficiency and budget, order-preserving, qualities of service including delay

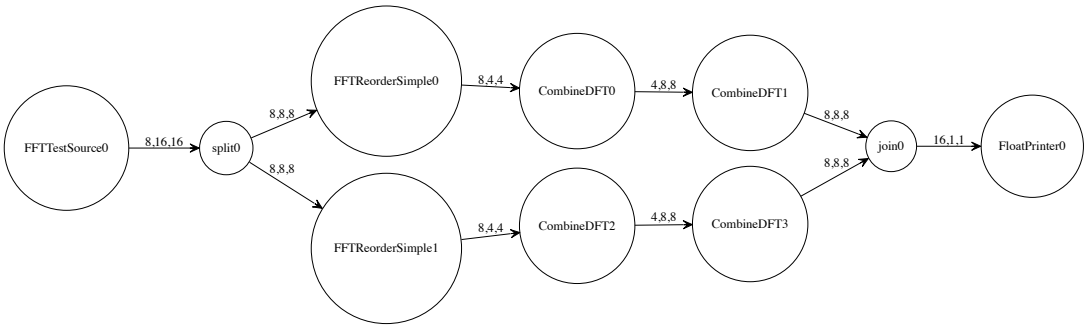


Figure 17: Coarse-grained FFT2.

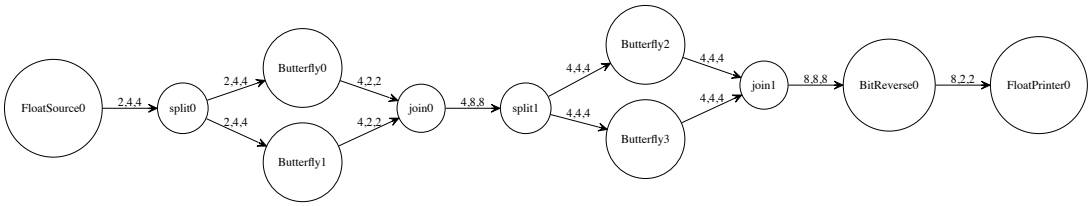


Figure 18: Fine-grained FFT3.

and accuracy. We believe there are opportunities in improving energy efficiency by automatically identifying the energy critical paths that consume a significant amount of energy and combining relevant stream optimizations to enhance the results. Even at a large scale of cloud streaming systems, DVFS techniques from the perspective of dataflows are likely to achieve significantly more energy savings in terms of computations rather than cluster management.

Though consistency and fault-tolerance in cloud computing seem to serve as extensions independent of existing stream optimizations, there might be inter-dependencies worth explorations. For example, super fine-grained partitioning and placement probably induce increasing overhead to maintain consistency and high availability such that the service level agreement (SLA) might not hold. An analytical model incorporated into stream compilers would be useful to combine the best of both worlds.