# Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs

Sumanaruban Rajadurai
School of Computing, National University of Singapore
sumanaruban@u.nus.edu

Jeffrey Bosboom
MIT CSAIL
jbosboom@csail.mit.edu

Weng-Fai Wong
School of Computing, National University of Singapore
wongwf@nus.edu.sg

Saman Amarasinghe
MIT CSAIL
saman@csail.mit.edu

## Abstract

An important class of applications computes on long-running or infinite streams of data, often with known fixed data rates. The latter is referred to as *synchronous data flow* (SDF) streams. These stream applications need to run on clusters or the cloud due to the high performance requirement. Further, they require live reconfiguration and reoptimization for various reasons such as hardware maintenance, elastic computation, or to respond to fluctuations in resources or application workload. However, reconfiguration and reoptimization without downtime while accurately preserving program state in a distributed environment is difficult.

In this paper, we introduce Gloss, a suite of compiler and runtime techniques for live reconfiguration of distributed stream programs. Gloss, for the first time, avoids periods of zero throughput during the reconfiguration of both stateless and stateful SDF based stream programs. Furthermore, unlike other systems, Gloss globally reoptimizes and completely recompiles the program during reconfiguration. This permits it to reoptimize the application for entirely new configurations that it may not have encountered before. All these Gloss operations happen in-situ, requiring no extra hardware resources. We show how Gloss allows stream programs to reconfigure and reoptimize with no downtime and minimal overhead, and demonstrate the wider applicability of it via a variety of experiments.

***CCS Concepts*** • **Software and its engineering → Just-in-time compilers**; **Dynamic compilers**; **Runtime environments**; *Distributed programming languages*;

***Keywords*** Downtime-Free Live Reconfiguration, Global Optimization, Distributed Compiler, Distributed Dynamic Recompilation, Cluster-wide Dynamic Recompilation, Program Migration, State Transfer, Computation Handover

**ACM Reference Format:**
Sumanaruban Rajadurai, Jeffrey Bosboom, Weng-Fai Wong, and Saman Amarasinghe. 2018. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173170

## 1 Introduction

Stream programs process large, even infinite, streams of data items through a stable but occasionally changing set of transformations. The *synchronous data flow* (SDF) [31] paradigm describes stream programs in which program units consume and produce a fixed number of data items during each activation. Important SDF-based streaming applications include cellular base stations, software-defined radios, radio astronomy applications, software defined mobile network functions such as in-network transcoding and Filter Bank Multi-Carrier (FBMC) [22], and virtualized network functions such as encryption/decryption, virus scanners, and deep packet inspection. The stable computation pattern of SDF-based stream programs allows compilers to perform the aggressive global optimizations necessary to keep up with high-rate input streams [8, 10, 11, 26, 27, 33, 44].

Since streaming applications can run for days, possibly indefinitely, there will inevitably be a need to *reconfigure* the system while these applications are running. Scenarios requiring this include hardware maintenance, or workload or resource fluctuations. In addition to mere reconfiguration, the ability to *recompile* a running application also enables global life-long reoptimization or online autotuning.

Recently several large scale SDF-based streaming applications are being considered for permanent cloud residency. Examples include the 5G standard proposals for software defined radio access networks (SD-RAN) and software defined core networks (SD-CN), which perform telecommunication signal processing entirely in the cloud or data centers [2, 3, 48]. Another example is the Square Kilometer

Array [19] project, which is exploring ways to move radio astronomy applications such as wide-field scientific imaging and beam forming [35] to the cloud [37, 45].

Such applications stand to reap various benefits (such as infrastructure sharing, elasticity, on-demand, and pay-as-you-go flexibility) if they are able to exploit highly dynamic hardware environments such as the elastic cloud or multi-tenant data centers. For this to happen, the compiler and runtime system must support *seamless live reconfiguration.*

Several *dynamic data flow* (DDF) [28] streaming systems support reconfiguration for elastic computing and fault tolerance [32, 36, 46, 51]. However, their techniques such as fine-grained checkpointing or input persisting introduce overhead during the program's normal execution [1, 38, 49] and *downtime* (time during which the program does not produce output) during reconfiguration [23]. Just like how garbage collection pause times were too disruptive for many applications before the development of concurrent collectors, the downtime of reconfiguration may prevent stream applications from meeting quality-of-service requirements.

After reconfiguration, some systems perform local recompilation, either explicitly or using an underlying layer such as a Java virtual machine (for example, Storm [46] or JESSICA2 [30, 52]). However, these recompilations are limited to locally reoptimizing at the granularity of the partitions used for load balancing. SDF stream programs, which are composed of fine-grained actors with predictable communication, allows for more powerful global optimizations such as actor fusion and automatic parallelization that are not possible for general programs. Such optimizations can have an order of magnitude performance impact [25], but their global nature makes them very sensitive to the contents of each partition. To have the option of full optimization across repeated recompilations requires a reconfiguration system flexible enough to globally repartition the program.

However, reconfiguration and recompilation of streaming computations is nontrivial. First, the program state (position and order of data streams and actor states) must be deterministically persisted and transfered to the new program instance. This is a challenging task in a distributed environment. Second, when optimizing for throughput, high-performance stream programs buffer data at intermediate points. The compiler needs to know the final state of the previous program instance to properly recompile the program for the new configuration, introducing a data dependency between the end of the old program instance and the start of the new program instance. Third, reconfiguration and recompilation itself takes time, which must be minimized and hidden, similar to modern garbage collectors. Finally, because reconfigurations are uncommon events, performance-impairing book-keeping should not be required during normal execution. At the same time, quality-of-service requirements dictate that the program should continue to produce output, and the transition between program instances should appear smooth to the client, with no downtime.

This paper describes the Gloss live reconfiguration system for SDF-based distributed stream programs. As far as we know, it is the first system that can seamlessly reconfigure and globally reoptimize SDF stream programs without downtime. Gloss is therefore not limited to a predefined set of configurations, giving it additional flexibility. Gloss introduces several techniques to hide the latency of reconfiguration and recompilation for both stateless and stateful stream programs. Moreover, these operations happen in-situ, requiring no extra hardware resources. To evaluate our work, we have implemented Gloss in StreamJIT [10], a just-in-time compiler and runtime system for distributed SDF programs. This paper makes the following contributions:

- We demonstrate the use of *cluster-wide dynamic recompilation* for stream programs that recompiles and redistributes programs on-the-fly in a distributed environment. To the best of our knowledge, Gloss is the first ever system that performs cluster-wide dynamic recompilation, thereby *global reoptimization.*
- We present a novel strategy, which we call *concurrent recompilation*, that recompiles a running program in two phases, reducing the visible recompilation time to sub-seconds and making cluster-wide dynamic recompilation practical.
- We present three live reconfiguration strategies that transfer the state of a stream program and resume the computation in a newly-compiled program instance in a distributed environment: *stop-and-copy reconfiguration*, *fixed seamless reconfiguration*, and *adaptive seamless reconfiguration.*
  - The last strategy completely eliminates the downtime during live reconfiguration.
- We propose a novel strategy called *asynchronous state transfer (AST)* that deterministically capture and transfer the state of a stateful program without checkpointing, input labeling, state recomputation or distributed consensus.
- We present two techniques for *output smoothing* that smooth the transition between old and new program instances that produce output at differing rates:
  - *Adaptive merging* avoids output rate spikes when transitioning to a faster new program instance by abandoning the old program instance as soon as the new program instance catches up.
  - *Resource throttling* reduces the compute resources available to a faster old program instance to allow the new program instance to catch up.

In the next section, we present a brief overview of the StreamJIT language and compiler. We then explain the details of Gloss and evaluate its effectiveness.

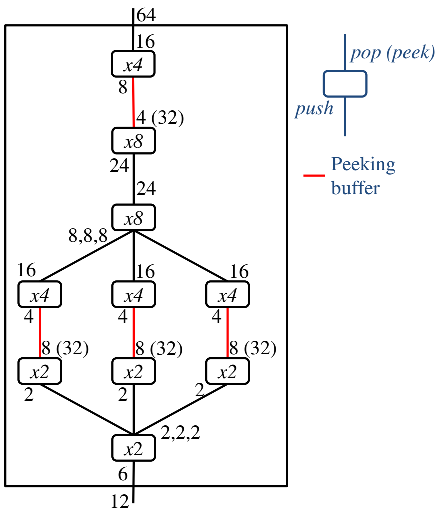


**Figure 1.** A stream graph, annotated with each worker's data rates. The *xN* in each worker denotes the number of executions of that worker in one possible schedule. In this schedule, the graph has overall pop rate 64 and push rate 12, i.e., it consumes 64 items and produces 12 items per execution of the schedule.

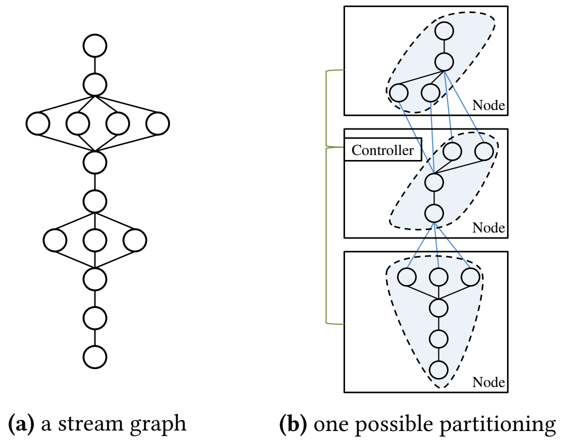


**Figure 2.** A stream graph partitioned into three blobs executing on different nodes. Besides the data channels between nodes, the controller node has a control channel to each other node (shown in green).

## 2 StreamJIT Language and Compiler

In this section we present a brief overview of StreamJIT [10], a Java-embedded programming language, compiler, and runtime system for SDF [31] paradigm stream programming.

StreamJIT programs are stream graphs (see Figure 1) composed from filters, splitters and joiners (collectively called *workers* as they all have work methods specifying their behavior). *Filters* are single-input, single-output workers[1]. *Splitters* and *joiners* have multiple outputs and inputs respectively. Both stateless and stateful workers are supported. A stream graph is stateless if all its workers are stateless.

All workers declare static *peek rates* stating how many items they examine on each input, *pop rates* stating how many of those items they consume, and *push rates* stating how many items they produce on each output for each firing. A *peek* operation is a read operation that does not remove data from the input. Peeking allows workers to perform sliding-window operations without using state variables. The StreamJIT runtime creates and maintains *peeking buffers* to store the required data items for peeking. Peeking is useful because stateless workers that use peeking remain stateless, preserving parallelization opportunities.

Fixed data rates enable StreamJIT to optimize for steady-state throughput. During compilation, StreamJIT applies domain-specific optimizations – such as fusing the workers together for locality, partitioning the graph for load balancing, removing built-in splitters and joiners to reduce memory traffic, and others [10] – and generates an execution schedule. As all communication between workers occurs via the stream graph edges, the StreamJIT compiler is free to select an execution schedule that exploits the appropriate mix of data, task and pipeline parallelism for a particular program and machine [25]. The fixed data rates allow the compiler to convert the logical queues on the graph edges to indexing on fixed-size circular buffers. By buffering sufficient data for each group of fused workers to execute in parallel, synchronization can be coarsened to a single barrier at which threads synchronize after executing their allocated portion of the execution schedule.

A StreamJIT program's lifecycle contains three phases: *initialization*, *steady-state*, and *draining*. The initialization phase executes a partial schedule to fill the buffers between workers. The program spends most of its execution time in the steady-state phase, an optimized multithreaded phase during which each thread alternates between executing its allocated portion of the execution schedule and synchronizing at a barrier. At the barrier, output is emitted and new input is ingested. When the input is exhausted or reconfiguration is requested, the draining phase attempts to flush the internal buffers, then returns the *program state* (the states of stateful workers and any remaining data items in the buffers) for use by the next configuration.

***Distributed StreamJIT Compiler*** StreamJIT's distributed runtime system partitions the stream graph into sets of connected workers called *blobs* (Figure 2). Each blob is compiled and executed independently. One node is designated as the *controller* that orchestrates the execution of blobs on all nodes, including itself.

At runtime, the controller partitions the graph into blobs and sends each blob to its host node, where it is compiled.

[1]Despite their name, filters need not remove items from the stream.

Each host node establishes network connections with the nodes hosting its upstream and downstream blobs to transfer data items, independent of the control connection to the controller node. Each blob executes independently (without global synchronization) as it receives input and produces output. Note that the StreamJIT compiler presented early in [10] does not recompile distributed stream programs.

## 3 Global Reoptimization

The complex execution environment (processor, memory system and network) of modern distributed systems leads to nonlinear interactions between compiler optimizations and changes in resources and workload. Runtime systems that perform load balancing by migrating predefined partitions between nodes necessarily sacrifice performance because there is no opportunity to do global optimization of all the partitions within a single node. The composition of smaller optimized partitions ('local optimal') is in general suboptimal. Large performance gains in stream programs often come from global whole program optimizations such as filter fusion, coarse grain parallalization with synchronization elimination, and load-balanced static work distribution between nodes [24, 26, 39]. Gloss dynamically recompiles and globally reoptimizes the stream programs using only the currently available resources during reconfiguration.

### 3.1 Dynamic Recompilation

Dynamic recompilation allows Gloss to alter any of its optimization decisions, including the amount of parallelism, internal data structures, distribution of computation over different cluster configurations and the execution schedule for a running program. For example, two filters that transfer large amounts of data between them can be assigned to the same partition, fused together and data-parallelized, optimizing for reduced network communication. However, if workload happens to increase to a point that the processor becomes the limiting factor for performance, the next recompilation can rearrange those filters into separate partitions and place those partitions onto separate nodes, possibly bringing new nodes into the program. In this way, Gloss's dynamic recompilation enables global load balancing without sacrificing optimization opportunities. These advanced optimizations are necessary to maintain the programs' quality-of-service in highly dynamic execution environments. To achieve this, the first challenge is state dependency, explained below.

***State Dependency*** The initial JIT compilation in StreamJIT is just like any other JIT compilation. However, recompilation requires the complete program state of the current program instance[2] (i.e., current running version). This is because some compiler optimizations such as fusion, unrolling, or built-in splitter and joiner removal [10] either alter the internal buffers or depend on the remaining data items in the internal buffers of the current program instance.

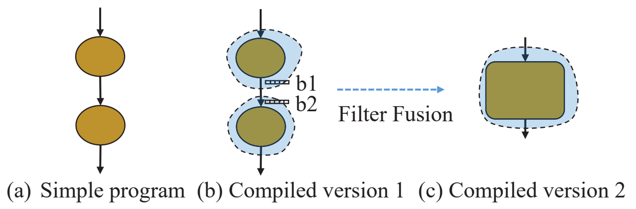


**Figure 3.** (a) A simple program where two filters are connected sequentially (b) The two filters were compiled separately and are running on two different machines. *b1* is the output buffer of the first filter, and *b2* is the input buffer of the second filter. (c) The filters are moved to a single machine and fused together.

For example, let's assume a simple stream program (shown in Figure 3) with two filters, where the filters were compiled separately and are running on two different machines. Suppose we want to bring the filters to a single machine and recompile the program. To decide whether to fuse the filters, the compiler needs to know if there will be any data items in between the filters in the current program state because the filters cannot be fused if such data exist. This is one domain specific optimization example where the initial data is required for the recompilation of a running program. Besides domain specific optimizations, the compiler needs to know the number of data items remaining in buffers after draining to compute the initialization and steady-state schedules and to specify how to adapt the old program instance's state so that computation can be resumed in the new program instance.

To avoid state dependency, one can always turn off such compiler optimizations during recompilation. However, this severely compromises performance. Therefore, we decided to keep all optimizations available during dynamic recompilation, including the tradeoffs between task, data and pipeline parallelism [25] within each blob. But state dependency creates a situation where the recompilation can take place only after the current version is stopped and the program state, which is distributed across nodes, is collected for the compiler, thereby incurring significant downtime. Before discussing how Gloss mitigates this downtime, we need to explain the reconfiguration process in Gloss because dynamic recompilation happens during reconfiguration.

## 4 Reconfiguration

Gloss can reconfigure a stream program for arbitrary new configurations. The most obvious reconfigurations includes changing how the graph is partitioned into blobs, moving a blob from one node to another, or the addition or deletion of computing nodes (see Figure 7). Gloss implements three

[2]Because StreamJIT programs are stream graphs, we call program instance as graph instance as well.

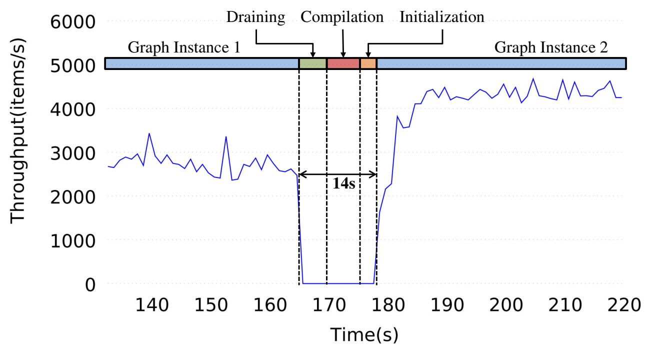


**Figure 4.** Time breakdown of stop-and-copy reconfiguration of Beamformer (a stateful stream program), increasing the number of nodes from two to three.

reconfiguration strategies of increasing sophistication: *stop-and-copy reconfiguration*, *fixed seamless reconfiguration*, and *adaptive seamless reconfiguration*. Note that all these reconfigurations use no extra hardware resources, and are performed on the original resources where the stream program is running. This section describes stop-and-copy reconfiguration. Section 7 will further describe the fixed and adaptive seamless reconfiguration strategies.

### 4.1 Stop-and-Copy Reconfiguration

To solve state dependency, the stop-and-copy reconfigurer stops the world during reconfiguration. First, the controller asks each blob to drain its portion of the stream graph. The controller waits until all blobs have returned any data items left in buffers after draining completes[3] and the state of any stateful workers in the blob (together called the *program state*). The controller rearranges the state to match the new configuration's partitioning of workers to blobs, then sends the new blobs and the corresponding state to their host nodes for recompilation, *which begins with complete program state*. The host nodes compile the blobs and install the buffered items and worker state, generating blobs that have the program state incorporated. We call these blobs *state-absorbed blobs*. The state-absorbed blobs continue execution from the old program instance's state.

The stop-and-copy reconfigurer enables all powerful domain specific optimizations to be applied to a running program, allowing the program to benefit across its entire lifetime. However, it results in long periods during which the stream program produces little or no output. This downtime has three major causes: *draining*, *recompilation*, and *initialization*. When draining begins, execution switches from the optimized, multithreaded compiled blob to the StreamJIT interpreter, which runs a fine-grained execution of the stream graph on a single thread to drain the buffered data, reducing throughput to near zero. Recompilation itself also takes time, during which the program is completely stopped. Finally, each blob's initialization phase usually requires receiving input from its upstream blobs, further delaying the downstream blobs' entry into their steady state.

Figure 4 shows the decrease in throughput during stop-and-copy reconfiguration. Draining, compiling, and initialization take five, six, and three seconds respectively, giving a total downtime of 14 seconds. While simple, stop-and-copy reconfiguration is not ideal for real-world stream applications with quality-of-service constraints, such as virtualized network functions or software defined radio access networks.

## 5 Concurrent Recompilation

This section explains Gloss's techniques to reduce the visible recompilation time that is the most significant of the three sources of downtime.

High recompilation time is one of the challenge that modern dynamically reoptimizing compilers face. For example, the Java HotSpot JVM [29], has two different JIT compilers: server and client JIT compilers. The client compiler performs only quicker compilations, which is suitable for interactive applications. The server compiler performs longer and aggressive compilation to ensure the program's performance, which is suitable for computation heavy applications.

As compiler optimization and code generation are expensive operations that take significant time, minimizing the compilation time to near zero is unlikely. Instead, the compilation time must be hidden by keeping the JIT recompilation process off the critical path. However, state dependency blocks us from doing so. To solve this problem, Gloss employs *implicit state transferring* for stateless programs and *two-phase compilation* for stateful programs, as will be explained below.

First, we categorize the stream programs into two classes: *stateful* and *stateless*. The stateful class includes the programs that contains one or more workers with state variables. The stateless class includes the rest, including the programs that perform peeking *even though the peeking buffers are stateful* (explained in Section 2).

The pure stateless programs have no state dependency issue. However, the state dependency issue exists in peeking stateless programs, where the peeking buffers need to be transferred from the old program instance to the new program instance when reconfiguring. Given that, we can avoid transferring peeking buffers between two program instances via input duplication and concurrent execution. That is, we can recompile stateless programs with no initial program state while the old graph instance is running, then run both graph instances concurrently, and fill the new peeking buffers using input duplication, thereby achieving an *implicit state transfer*. The runtime system can handle this, and no compiler changes are needed. Section 6.1 explains

[3]Because workers execute atomically, draining cannot completely empty a buffer whose size is not divisible by the downstream worker's pop rate, or if the downstream worker peeks.

the input duplication and concurrent execution techniques, and Section 7.1.1 explains the reconfiguration strategy for stateless programs.

Stateful programs, however, require explicit state transfer. Both the state of variables, and buffered data have to be transferred before the start of recompilation. The concurrent recompilation strategy solves this state dependency via a two phase compilation strategy, explained in the next subsection.

### 5.1 Two Phase Compilation

For stateful programs, Gloss recompiles in two phases. The first phase happens concurrently while the old graph instance is running. To start the process, instead of draining the old graph (as in stop-and-copy reconfiguration), the runtime system collects the steady-state buffer capacities (called the *meta program state*) from the old program instance, and gives it to the compiler.

Gloss then performs optimizations that do not require the program state as well as those that do, the latter using the meta program state. Specifically, it recompiles the blobs as in the serial recompilation. However, it will merely mark out of operations that need actual program state, namely (1) splitter and joiner removal, and (2) the generation of the read instructions of initialization schedule. This phase completes all compiler operations except these two, hence this is a heavy compilation phase. Finally, it generates a blob, called a *pseudo-blob*, that is compiled but not runnable as it needs actual program state to be injected.

The second phase takes place after the old program instance is stopped (drained) and the program state is collected. Once the program state is collected from the old graph instance, the runtime system passes the state to the compiler. The compiler then performs the marked operations mentioned above and installs the program state, converting pseudo blobs to state-absorbed blobs, which is ready for execution.

In summary, the first phase, which is a heavy compilation, does all the time-consuming compiler operations, generating a *pseudo-blob*. The second phase, which is a light compilation, converts the pseudo blob to a state-absorbed blob. Figure 5 depicts the time breakdown of the two-phase recompilation strategy. The first phase happens in the background while the old program instance is running, and thus it is hidden. The second phase happens just-in-time. Effectively, two phase recompilation strategy brings the visible recompilation time to sub-seconds.

## 6 Runtime Techniques Enabling Zero Downtime Reconfiguration

As shown in Figure 5, we still have downtime, which comes mainly from the draining and initialization phases. In this section, we explain two runtime techniques that overcome the remaining downtime.

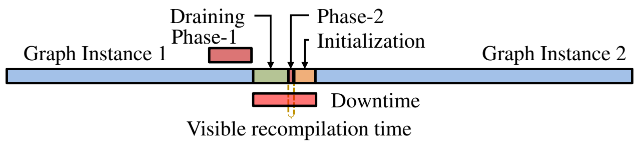


**Figure 5.** Time breakdown of the two-phase recompilation strategy.The phase-1 compilation happens while the graph instance-1 is running. Only the phase-2 becomes visible and contributes to the downtime.

### 6.1 Input Duplication and Concurrent Execution

The initialization phase of a blob's lifecycle fills the internal buffers for use by the steady-state phase (see Section 2). Internal buffering allows fine-grained synchronization between workers to be replaced by a single global barrier at which all of the blob's threads synchronize after each execution of the steady-state schedule. Coarsening synchronization and amortizing it across a full schedule is crucial to achieving high throughput. The draining phase performs the opposite, flushing the internal buffers and sending any remaining data to the controller.

Initialization is inherently sequential, as upstream workers must execute to fill buffers before downstream workers can begin execution. Similarly, draining is also inherently sequential, as upstream workers must drain their buffers before downstream workers can begin draining. While the steady-state phase is optimized and multithreaded, the initialization and draining phases are single-threaded, hence have lower throughput.

Because SDF stream programs are deterministic, Gloss hides this downtime by duplicating the input and running both the old and new graph instances concurrently, ensuring the stream program continues to produce output at the cost of some redundant work.

### 6.2 Zero-Overhead Program State Capture via Asynchronous State Transfer

In order to perform concurrent execution, reconfiguration requires the ability to deterministically persist the state of a program running across multiple nodes, and pass it to the new program instance. This is challenging because the program state is continuously changing and distributed across multiple nodes.

Besides stop-and-copy, another simple reconfiguration strategy is to record periodic checkpoints of the program state at well-defined points in its execution and persist the inputs [1, 15, 32, 36, 46]. Reconfiguration can then revert to the state captured at the previous checkpoint and process the persisted input, albeit losing the work performed after the checkpoint. Unfortunately, this introduces not only downtime but also overhead during normal execution.

Gloss's solution is *asynchronous state transfer (AST)*, in which the controller requests each of the old graph instance's

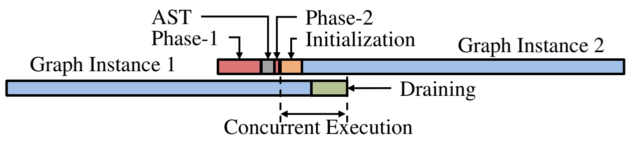


**Figure 6.** Time breakdown of Gloss's techniques.

blobs to send a copy of its state after processing the *n*th data item in the stream, where the *n*th data item is predicted to be consumed by the graph *t* seconds in the future. Because this request happens via the control channel while the input data is being consumed by the graph via the data channel, *n* must be far enough into the future to ensure the blobs receive the request before processing item *n*. *n* is calculated by the controller based on the overall graph's input consumption rate and a constant *t* (we use 3 seconds), allowing for network delay in receiving the request. For each blob, the number of data items consumed in each execution can be computed from the schedules and the statically-known data rates of the workers, so counting the number of items processed by the blob requires only one addition instruction per schedule and does not require labeling the data items.

Once the designated number of items have been processed, when the blob's threads next reach its barrier (see Section 2), one thread copies worker state and buffered data items (blob's program state) and sends them to the controller. After copying, the blob continues execution as normal without the blob entering the draining phase. The controller then builds the program state of the whole graph instance by merging the buffered items it receives.

AST captures a valid state of a program while the program is running in a distributed environment, without incurring overhead during normal execution, i.e., no checkpointing, input labeling, state recomputation or distributed consensus. Although the distributed state is collected in the controller for state rebuilding, it is done outside the critical path. The program can continue to produce output even after state transfer, effectively avoiding downtime.

Figure 6 depicts the breakdown of the Gloss. Reconfiguration begins with the phase-1 of the concurrent recompilation strategy, and AST takes place just after phase-1. Phase-2 happens immediately after AST. Following that concurrent execution and input duplication occurs.

## 7 Seamless Reconfiguration

This section presents our *fixed* and *adaptive* seamless reconfiguration strategies. Both seamless reconfiguration schemes use concurrent recompilation and the techniques from Section 6, namely asynchronous state transfer, concurrent execution, and input duplication. The two schemes differ in the way they switch between the old and new graph instances. In the fixed seamless reconfiguration scheme, a transition point is computed in advance using the schedules and data rates.

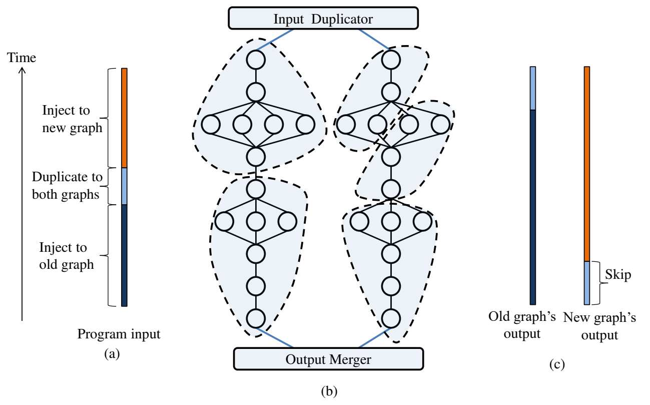


**Figure 7.** Input duplication during fixed seamless reconfiguration, moving from 2 blobs to 3 blobs. (a) Around the switching point, some of the input is duplicated and fed to both the old and new graph instances. (b) Both graph instances execute concurrently. (c) The old graph instance's output, including the output based on the duplicated input, is used. The beginning of the new graph instance's output (based on the duplicated input) is skipped and the remainder is used. The colors in (a) and (c) depict input and its corresponding output.

The new graph instance's output stream takes over after the old graph instance stops. Fixed seamless reconfiguration is simple, and bounds the amount of redundant work performed. But it may exhibit downtime or output rate spikes if the old and new graph instances run at different speeds. Adaptive seamless reconfiguration dynamically transitions between the old and new graph instances when the new graph instance catches up, smoothing the output rate during the transition.

### 7.1 Fixed Seamless Reconfiguration

As shown in Figure 7, fixed seamless reconfiguration runs the old and new graph instances concurrently on a fixed amount of duplicated input. The old graph instance is stopped after processing the duplicated input, but the new graph instance continues to execute. All output produced by the old graph instance before it stops is forwarded to the graph's output. The portion of the new graph instance's output corresponding to the duplicated input is discarded, after which the new graph instance's output becomes the blob's output.

The amount of duplicated input is computed based on the data rates of the workers in the stream graph and the execution schedule of the blob. The computation is different for stateless and stateful graphs, as described in the next subsections.

#### 7.1.1 Stateless Graph

Stateless graphs do not contain any workers with state variables, but they still require internal buffering for both correctness (to implement peeking/sliding window operations) and performance (amortizing the cost of synchronization over many data items). Buffered data items must be transferred from the old to new graph instance during reconfiguration. Because stateless graphs have no worker state, we can *implicitly transfer* the buffered data items by duplicating input, leaving the new graph instance with the same buffered items as the old graph instance without stopping the old graph instance to copy the items.

Let $G_{in}^{init}$ denote the number of input data items needed to fill the internal buffers of a graph $G$ during its initialization phase. Similarly, let $G_{in}^{steady}$ denote the number of data items required during one execution of $G$'s steady-state schedule. $G_{in}^{init}$ and $G_{in}^{steady}$ are computed by multiplying the topmost worker's pop rate by the number of executions of that worker specified in the initialization or steady-state schedule respectively. Fixed seamless reconfiguration runs the old and new graph instances concurrently by duplicating $X \times \text{OLD}_{in}^{steady}$ data items, where

$$X = \left\lceil \frac{\max(\text{OLD}_{in}^{init}, \text{NEW}_{in}^{init})}{\text{OLD}_{in}^{steady}} \right\rceil$$

The use of $\max(\text{OLD}_{in}^{init}, \text{NEW}_{in}^{init})$ ensures that after $X$ executions of the old graph instance's steady-state schedule, any data items buffered in the old graph instance have been fully processed, and that the new graph instance has received enough (duplicate) input to complete its initialization schedule. The number of redundant output data items produced is $X \times \text{OLD}_{out}^{steady}$, one copy of which must be discarded. Because the new graph instance's initialization phase usually has a lower output rate than the old graph instance's steady-state phase, we forward all of the old graph instance's output and discard the first $X \times \text{OLD}_{out}^{steady}$ items produced by the new graph instance. At this point, reconfiguration is complete and the new graph instance executes normally.

### 7.1.2 Stateful Graph

Stateful stream graphs contain workers with state variables whose values must be transferred from the old to new graph instance, preventing implicit state transfer. Instead, Gloss uses AST (Section 6.2) to capture the state (both worker state variables and buffered data items) of the old graph instance. Gloss also uses two phase compilation strategy (Section 5.1) to recompile the new graph instance. Figure 6 depicts the reconfiguration and recompilation process of stateful graph.

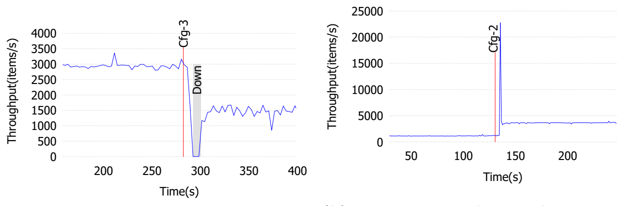
The intended buffer sizes of the old and new graph instances may be different, so the new graph instance's initialization schedule shrinks or enlarges the buffers as required. To allow the initialization schedule to complete, $X \times \text{OLD}_{in}^{steady}$ data items are duplicated, where

$$X = \left\lceil \frac{\text{NEW}_{in}^{init}}{\text{OLD}_{in}^{steady}} \right\rceil$$

As in the stateless case, $X \times \text{OLD}_{out}^{steady}$ redundant output items are skipped from the next graph instance's output buffer.

### 7.1.3 Issues in Fixed Seamless Reconfiguration

When the new graph instance has a slightly higher throughput than the old, fixed seamless reconfiguration will result in zero downtime. A slightly higher throughput is required for the new graph instance to catch up with the old one because the new graph instance need to amortize its initialization phase, which is slower. In other cases, however, fixed seamless reconfiguration either causes downtime or spikes in the output rate during the transition.



**(a)** Downtime when transitioning from high to low throughput.

**(b)** Output spikes when transitioning from low to high throughput.

**Figure 8.** Downtime and output spikes during fixed seamless reconfiguration when throughput differs.

When the old graph instance has a higher throughput than the new graph instance, the old graph instance processes all of its input and stops before the new graph instance completes its initialization schedule, resulting in downtime. The downtime is shorter than that caused by stop-and-copy reconfiguration, but still long enough to be disruptive. Figure 8a shows the downtime caused by fixed seamless reconfiguration from high to low throughput.

When the old graph instance has a lower throughput than the new graph instance, the new graph instance can buffer up lots of output while waiting for the old graph instance to finish processing its input. When the old graph instance finally stops, this buffered output results in a spike in the output rate, which can disrupt clients of the stream application. Figure 8b shows output spikes caused by fixed seamless reconfiguration from low to high throughput. In theory, downtime and output spikes can be avoided by predicting the throughput of the new configuration, and adjusting the fixed transition point between the old and new graph instances. In practice, building a robust throughput predictor is difficult because performance depends on (possibly rapidly) changing runtime factors as well as static features of the configuration. In the next section, we explain how our adaptive seamless reconfiguration scheme addresses the remaining downtime and output spikes by deciding the transition point dynamically.

### 7.2 Adaptive Seamless Reconfiguration

Like the fixed scheme, adaptive seamless reconfiguration also runs the old and new graph instances concurrently, but instead of duplicating a fixed amount of input, adaptive seamless reconfiguration dynamically chooses how much input to duplicate and thus when to switch from the old to the new graph instance. Adaptive seamless reconfiguration applies two output smoothing techniques, *adaptive merging* and *resource throttling*, to avoid the downtime and output rate spikes of fixed seamless reconfiguration.

***Adaptive merging*** Output spikes are avoided by switching to the new graph instance as soon as it catches up with the old graph instance, avoiding the buildup of buffered output that causes spikes. How much input to duplicate is computed in the same way as in the fixed seamless reconfiguration scheme, but if the new graph instance catches up before the old graph instance has finished processing all the duplicated input, the old graph instance is immediately abandoned. Input duplication results in redundant outputs. In the fixed seamless reconfiguration always discards the first outputs of the new graph instance. However, in adaptive merging output from the old, new or both instances may be discarded, depending on when the new graph instance catches up.

***Resource throttling*** The adaptive seamless reconfiguration scheme avoids the downtime incurred when the new graph instance has lower throughput than the old graph instance by resource throttling, effectively slowing down the old graph instance so the new graph instance can catch up. Based on the time elapsed since the reconfiguration process started, the number of cores allocated to the old graph instance is repeatedly halved, resulting in the old graph instance having more threads than cores available. Timeslicing reduces the old graph instance's throughput, while simultaneously freeing more resources for the new graph instance, increasing its throughput. If this does not slow the old graph instance sufficiently, the old graph instance's input rate is further restricted.

### 7.3 Summary

The flowchart in Figure 9 summarizes adaptive seamless reconfiguration, which consists of the following steps:
***Compilation.*** When reconfiguration begins, Gloss compiles the new graph instance while the old graph instance continues to execute.
***State transfer.*** If the program is stateful, the state of the old graph instance is collected via asynchronous state transfer.
***Input duplication.*** After injecting the collected state (if any), the new graph instance begins executing. Input data items are sent to both the old and new graph instances. The old graph instance's output continues to be forwarded to the overall blob output.

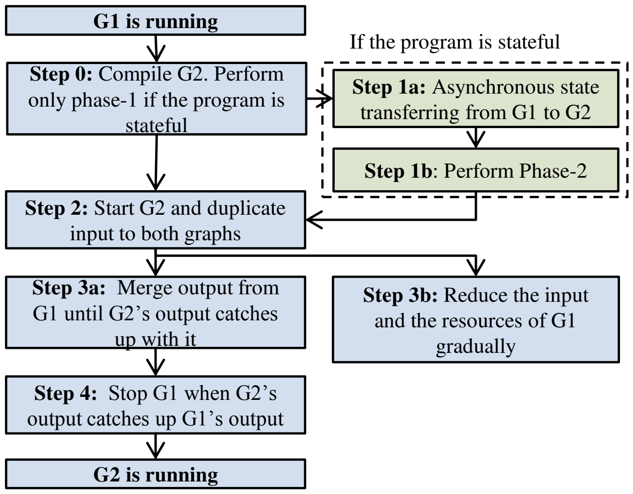

G1 is running
If the program is stateful
Step 0: Compile G2. Perform only phase-1 if the program is stateful
Step 1a: Asynchronous state transferring from G1 to G2
Step 1b: Perform Phase-2
Step 2: Start G2 and duplicate input to both graphs
Step 3a: Merge output from G1 until G2's output catches up with it
Step 3b: Reduce the input and the resources of G1 gradually
Step 4: Stop G1 when G2's output catches up G1's output
G2 is running

**Figure 9.** Flowchart of adaptive seamless reconfiguration from old graph instance G1 to new graph instance G2.

***Resource throttling.*** Even as the old graph continues to execute, its execution resources and input rate are gradually reduced, thereby reducing its throughput so that the new graph instance can catch up.
***Adaptive merging.*** Once the new graph instance has caught up with the old graph instance, the old graph instance is stopped, and redundant output are discarded.

## 8 StreamJIT Applications

Before discussing the evaluation of Gloss, we shall briefly explain two applications that were implemented in StreamJIT. The applications are stateless, and were used to evaluate how Gloss can be useful to migrate a single-node program entirely from its host node to a new node (Section 9.8). In the real world, we can expect many parallel instances of these applications running in the system.

### 8.1 LTE-A Uplink Transceiver

4G LTE-A (Long Term Evolution-Advanced) is a standard for wireless communication of high-speed data (up to 1 Gbps) for mobile phones and data terminals. It incorporates several techniques such as carrier aggregation, high-order MIMOs, high-order QAM, and Turbo decoding.

The LTE-A Uplink Transceiver implemented in StreamJIT includes an uplink transmitter, a MIMO channel, and an uplink receiver. The program implements a $2 \times 2$ MIMO channel with spatial multiplexing. The transmitter comprises the turbo encoder, outer interleaver, 64 QAM modulator, FFT, mapper, and IFFT filters. The receiver is made up of the subcarrier demapper, MIMO equalizer, demodulator, outer deinterleaver and turbo decoder filters. This application is the core of any LTE-A base station implemented in the cloud.

| Application | State | Stop and Copy | | Fixed | | Adaptive | |
|---|---|---|---|---|---|---|---|
| | | Throughput disrupted time (s) | Down time (s) | Throughput disrupted time (s) | Down time (s) | Throughput disrupted time (s) | Down time (s) |
| Beam Former | Stateful | 10.7 | 9.3 | 5.5 | 1.5 | 7.0 | 0 |
| Vocoder | Stateful | 15.2 | 11.8 | 2.5 | 0.75 | 6.0 | 0 |
| TDE_PP | Stateless | 9.0 | 8.4 | 6.4 | 4.8 | 3.83 | 0 |
| FMRadio | Stateless | 10.5 | 9.2 | 9.5 | 1.0 | 7.0 | 0 |
| SAR | Stateless | 9.8 | 6.2 | 5.2 | 1.3 | 2.1 | 0 |
| Filter Bank | Stateless | 7.8 | 6.16 | 3.85 | 2.28 | 2.8 | 0 |
| **Average** | | **10.50** | **8.51** | **5.49** | **1.92** | **4.78** | **0** |

**Table 1.** Average throughput disrupted time and average downtime of Gloss's reconfiguration schemes.

### 8.2 DVB-T2 Receiver

DVB-T2 (Digital Video Broadcasting-Terrestrial 2) is the latest digital terrestrial television standard, targeting not only fixed antennas, but also PCs, laptops, portable radios, smart phones, dongles, and a whole range of other portable devices. Compared to the previous technology (DVB-T), it improves spectral efficiency, increases throughput, simplifies transmitter and receiver, and supports HD, SD, and UHD transmissions.

The DVB-T2 receiver implemented in StreamJIT comprises the FFT, Channel Estimator, Frequency Deinterleaver, Cell Deinterleaver, Constellation Derotation, Forward Error Correction, Frame Multiplexer, Bit Deinterleaver, and LDPC (low density parity check) decoder filters.

## 9 Evaluation

In this section, we first study the reconfiguration schemes by comparing their relative contribution to reducing downtime, followed by an evaluation of the overhead of the better scheme, i.e., adaptive seamless reconfiguration. We then show how Gloss's seamless reconfiguration can be useful in various scenarios. Following that, we investigate the effect of program state size on reconfiguration time. Finally, we show how Gloss performs with the two real-world applications described in Section 8. We use scaled up versions of the original benchmark applications from StreamJIT, a comprehensive summary of which can be found in [24]. Except for the experiments reported in Sections 9.3 and 9.4, the experiments were performed using identical nodes equipped with dual-socket 2.4GHz Intel Xeon E5-2695v2 processors (12 cores per socket, 2 threads per core) with 128GB RAM and Mellanox ConnectX-3 EN 10GbE NICs running Ubuntu 14.10, with StreamJIT running on OpenJDK 1.8u31.

As part of StreamJIT's throughput optimizations (described in Section 2), all StreamJIT programs ingest input and emit output at the end of each execution of the steady-state schedule. In other words, a program will ingest $I$ at the beginning of the cycle, then output $O$ at the end, leaving a very short period (in milliseconds) where no output can be seen. To account for this, in this section and throughout the paper, we measure throughput at the granularity of one second.

### 9.1 Comparison

Table 1 compares the average length of time in which throughput was disrupted, as well as the average downtime of Gloss's reconfiguration schemes for six different applications. Each application was reconfigured 100 times using the respective scheme on eight nodes, and the average was computed. We use the same 100 different configurations to test the reconfiguration schemes. The last row summarizes the overall average of the schemes. By *downtime*, we mean a significant period of time in which the stream program is not producing any output. *Throughput-disrupted time* is period of time during which the stream program is producing less than what it averages in the previous 100 seconds. We refer to the latter as *full throughput*.

Stop-and-copy reconfiguration suffers from significant downtime and throughput-disrupted time as explained in Section 4.1. Fixed seamless reconfiguration reduces the average downtime, but it fails to totally eliminate it due to the reasons described in Section 7.1.3. Adaptive seamless reconfiguration completely eliminated downtime with significantly lower throughput-disrupted times.

### 9.2 Reconfiguration Overhead

To measure the overhead of adaptive seamless reconfiguration, we repeatedly reconfigured FMRadio into the same configuration. Using the same configuration ensures that the measured reduction in throughput is due to reconfiguration, not the properties of the new configuration. Figure 10 shows the throughput of FMRadio running on 8 nodes through 3 reconfigurations. The shaded region shows when the old and new graph instances are running concurrently. On average, the two graph instances overlapped for 7.2 seconds. The program does not experience any downtime though throughput is reduced by 27% during the reconfiguration process.

Recompilation and reconfiguration do not use extra resources. They run on the same resources where the original

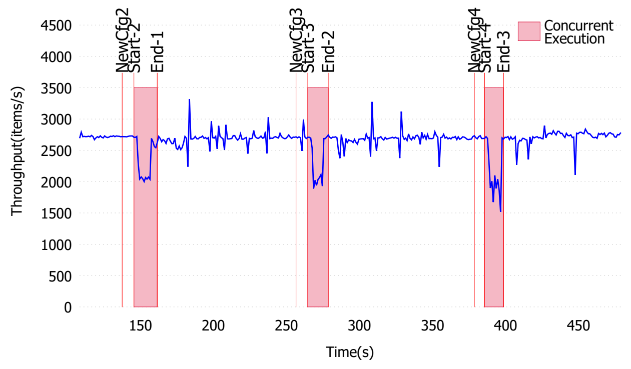
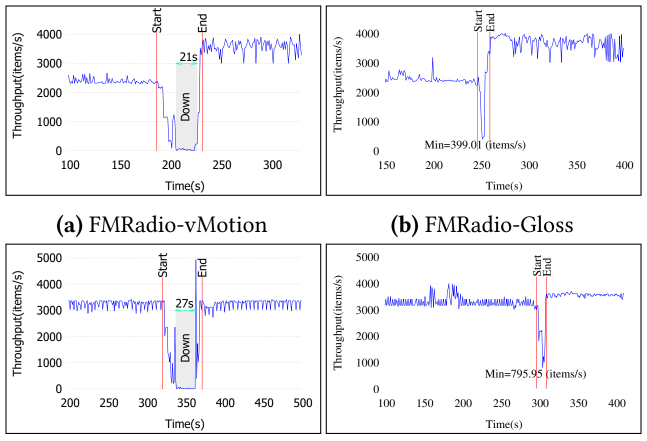


**Figure 10.** Throughput of FMRadio (a stateless stream program) running on 8 nodes during three adaptive seamless reconfigurations into the same configuration. The lines marked *NewCfg* denote the beginning of reconfiguration; the shaded area indicates when the old and new graph instances are running concurrently.



**(a)** FMRadio-vMotion **(b)** FMRadio-Gloss

**(c)** Beamformer-vMotion **(d)** Beamformer-Gloss

**Figure 11.** Downtime comparison of vMotion migration and Gloss adaptive seamless reconfiguration for FMRadio (a stateless program) and Beamformer (a stateful program).

program is already using (via normal OS scheduling). This is also a contributor for the dip in throughput.

## 9.3 Gloss vs. VM Migration

An application-independent (and agnostic) approach to reconfiguration is virtual machine live migration. This is the de facto standard in cloud computing, even for large scale SDF applications [16].

We compared vMotion [34] migration with adaptive seamless reconfiguration by running stream programs inside virtual machines, and measuring the throughput during migration and reconfiguration. The experiment used vSphere 5.0.0, 64-bit CentOS 7.7, and Oracle JDK 1.8u31. The virtual machines were configured with 6 virtual cores and 32GB RAM. The physical hosts were equipped with 2.3GHz Intel Xeon E5-2630 processors and 32GB RAM. Each stream program initially runs on two nodes, then the second node is moved to a new physical host by either migration or reconfiguration.

As shown in Figure 11, vMotion migration had more downtime than adaptive seamless reconfiguration. vMotion first speculatively copies the contents of the virtual machine's memory, on the assumption that most memory pages will not change before the VM is migrated. This assumption does not hold for stream applications as they continuously ingest new data, modifying memory significantly. When vMotion detects that the amount of memory to be copied is not decreasing (because memory is being modified faster than it can be copied), it artificially slows down the virtual machine, a strategy named *stun during page send* [40]. Eventually, vMotion stops the VM and copies the remaining modified memory, but not before a lengthy throughput-disrupted time.

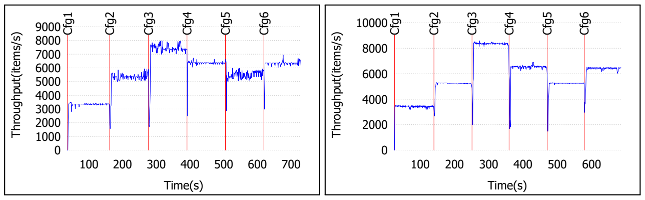
In many application scenarios, especially those involving live streams, this is not acceptable. For example, using techniques like forward error correction, a cellular or Internet call can tolerate brief (subsecond level, say) throughput dips or downtime with little impact on the user. These applications can cope with some reduced throughput by reducing audio or video quality, the user impact of which is proportional to the reduced-throughput time. But the multi-second pause times introduced by vMotion migration would result in dropped calls, which are very disruptive to the user. Gloss minimizes the impact of reconfiguration by making just one copy of the state using asynchronous state transfer, allowing the stream program to run at full speed before and after the copy, only throttling the old graph instance's resources when necessary for the new graph instance to catch up.

## 9.4 Elastic Computing

We evaluated Gloss's elastic computing capability by running stream programs on Amazon EC2, and then dynamically adding and removing nodes. The virtual machines used in these experiments all had 16vCPU and 32GB RAM, running 64-bit Ubuntu 15.04 and 64-bit Oracle JDK 1.8u31. Figure 12 shows the throughput of two programs as nodes are added and removed using adaptive seamless reconfiguration. Gloss reconfigures the programs without introducing any downtime, allowing seamless scaling out or in. Gloss also potentially allows cloud-based streaming program to exploit *spot instances* [4] to minimize execution cost.

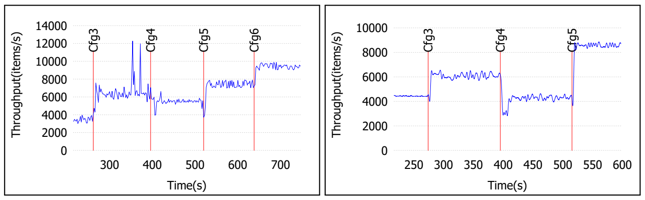
## 9.5 Online Autotuning

Program autotuning optimizes programs by running them many times in different configurations, using techniques from mathematical optimization and machine learning to guide the search for the best configuration. Gloss can dynamically recompile and seamlessly reconfigure programs from any point in the optimization space to any other point,



**(a)** Beamformer **(b)** FMRadio

**Figure 12.** Throughput of Beamformer (a stateful stream program) and FMRadio (a stateless stream program) on Amazon EC2 as nodes are added and removed. Programs initially run on two nodes, then two nodes are added, two more nodes are added, one node is removed, another node is removed, and one node is added. Gloss reconfigures the programs with zero downtime.



**(a)** Beamformer **(b)** FMRadio

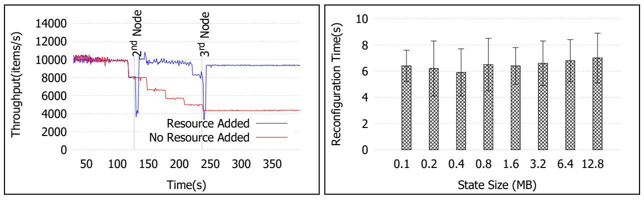
**Figure 13.** An excerpt of online autotuning of Beamformer (a stateful stream program) and FMRadio (a stateless stream program) on 8 nodes. Gloss reconfigures the programs with zero downtime. Throughput varies as the autotuner tries different program variants.

allowing global reoptimization unconstrained by earlier compilation decisions. This enables online autotuning on production data by eliminating pauses when transitioning between configurations, allowing the program to perform useful work during the tuning process.

Figure 13 shows a small excerpt of online autotuning of two stream programs, each on eight nodes. Throughput varies as the autotuner tries different program variants, but Gloss seamlessly reconfigures the program from one variant to another while maintaining quality-of-service.

### 9.6 Workload Fluctuation

Gloss achieves performance resiliency by continuously recompiling and reoptimizing the program in response to environment changes. To demonstrate this, we created a benchmark that increases the work required to process each data item every 30 seconds, starting after 100 seconds. The program initially runs on a single node, with a new node added whenever the throughput falls below 8,000 items processed per second. Figure 14a compares the throughput on this benchmark with and without adding additional nodes. Without adding nodes, the program can only achieve slightly more than half of the desired performance level. However, when using reconfiguration to add nodes, the program successfully maintains throughput with only a small amount of throughput-disrupted time.



**(a)** **(b)**

**Figure 14.** (a) Throughput as workload increases every 30 seconds beginning at 100 seconds. The blue line shows throughput when a node is added when throughput drops below 8000 items per second; the red line shows throughput of the same workload when nodes are not added. (b) Average reconfiguration time for different program state sizes.

### 9.7 Reconfiguration Time vs State Size

To evaluate the effect of program state size on adaptive seamless reconfiguration, we created a benchmark of different state sizes on the 8 nodes. Figure 14b shows the average reconfiguration time of 100 reconfigurations for each state size. Although Gloss collects the state in the controller to rebuild the complete state, it is done off the critical path through asynchronous state transfer that requires no global synchronization. The results validated this and showed that the size of the program state does not significantly affect reconfiguration time.

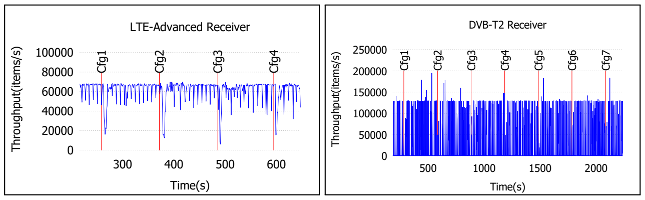
### 9.8 Single Node Experiment (Full Program Migration)

We ran LTE-A receiver and DVB-T2 receiver on a single node, and from time to time, the programs were migrated to a new machine. Figure 15a shows the throughput of LTE-A receiver, where the program is migrated from its current node to a new node repetitively for four times with the help of Gloss. Figure 15b shows the throughput of DVB-T2 receiver for the similar experiment. DVB-T2 receiver produces output in burst for every 2 seconds because of its high peek and pop rates, which is inherent to the application.

As shown in the figures, Gloss migrates full program from its host node to a new node with no downtime. Gloss empirically demonstrates the important benefits of software defined radio (SDR), which are hardware independency and flexibility, with these real world applications.

## 10 Related Work

***VM live migration*** Virtual machine live migration [14, 34] is an application-independent way to move workloads from one physical host to another for load balancing or to facilitate



**(a)** LTE-A Receiver **(b)** DVB-T2 Receiver

**Figure 15.** Throughputs of two programs (LTE-A Receiver and DVB-T2 Receiver), initially running on a single node. Time-to time, the programs were completely moved from there host node to a new node.The lines marked *Cfgn* denote the beginning of reconfiguration *n*.

maintenance. As shown in Section 9.3, VM live migration is not suitable for stream programs.

***Distributed database migration*** Recent work [17, 20, 21, 42, 47] focuses on live OLTP workload migration in distributed databases for load balancing and elastic scaling. Albatross [17] migrates active OLTP workloads of a shared-storage DBMS by iteratively copying the database cache between nodes. Squall [20] migrates data tuples between nodes of a distributed in-memory database while maintaining ACID semantics. Squall maintains responsiveness by first migrating tuples being accessed by transactions. DrTM+B [47] exploits existing fault-tolerant mechanisms, which maintain data replicas, to achieve efficient live workload migration.

***Distributed stream processing*** Distributed stream processing systems [1, 5, 9, 15, 36, 38, 46, 49–51] support reconfiguration for fault tolerance, dynamic scaling, and load balancing, but their strategies all incur overhead during normal execution. Spark Streaming [51] tracks the lineage of each dataset (the operations performed to produce them) and recomputes from occasional checkpoints. Matteis et al. [18] present a replica based strategy for low-latency reconfiguration. Apache Storm [6, 46] and MillWheel [1] use fine-grained checkpointing, input persisting, and acknowledgment; the receiver acknowledges every data item received from the sender. Using the fixed data rates of stream programs, Gloss does not require checkpointing, input persisting, or acknowledgment, avoiding any overhead during normal execution. Furthermore, Gloss's global recompilation enables load-balancing and scaling without sacrificing optimization opportunities.

DDF stream processing systems such as Flink [5], Spark Streaming [51], and Storm [46] lacks some essential SDF constructs. This also prevents the use of SDF specific optimizations, making them at least an order of magnitude slower in executing SDF code compared to Gloss/StreamJIT.

Erlang [7] is a domain-specific language for soft real-time telecommunications programming. To allow software upgrades without service disruption, Erlang supports hot swapping of program code via load balancing. New sessions run the new code, then after all old sessions have expired, the old code is unloaded. Flux [41] profiles communication rates between operators and performs fine-grained load balancing by moving individual operators; this fine-grained approach sacrifices opportunities for operator fusion. Gedik et al. [23] presents elastic scaling techniques for stream programs. However, the vertical and horizontal barriers across multiple nodes and the paused data stream at the splitter during the migration phase incur significant downtime. Siyoum et al. [43] analyze if an SDF program runs on a MPSoC can meet QoS requirements during graph level dynamism. We do not address graph level dynamism at the moment.

Cloud computing frameworks such as EventWave [13] and Orleans [12] support state migration for load balancing, but they use fixed partitions, and do not reoptimization.

## 11 Conclusion

Runtime reconfiguration is a practical necessity for long-running stream programs. We describe, for the first time, compiler and runtime techniques that enable downtime-free reconfiguration of SDF-based stream programs. Unlike other systems based on load balancing of fixed partitions, Gloss's seamless reconfiguration enables load balancing and continuous global reoptimization. Compared to the tens of seconds of no output during migration using a state-of-the-art commercial virtualization tool, Gloss can reconfigure stream programs with no downtime, additional resources, or overhead during normal execution. It is also able to respond to demand changes, allow for elastic scaling in the cloud, as well as make online autotuning feasible.

The cluster-wide dynamic recompilation makes all ahead-of-time domain specific optimizations available to a running program. The downtime-free live reconfiguration globally reoptimizes and redistributes program instances across cluster nodes on-the-fly.

Gloss solves a critical problem before large scale SDF programs can take advantage of the elastic cloud. For example, Gloss can help in the implementation of the Cloud Radio Access Network (Cloud-RAN or RANaaS) concept in 5G mobile network where RANs are to be migrated from expensive and rigid custom hardware to software platforms running on the elastic cloud. In summary, Gloss enables cost-effective, flexible and responsive *commoditization* and *virtualization* of large and complex SDF-based stream programs.

## Acknowledgments

The work reported here was supported by Singapore Ministry of Education under Academic Research Fund Tier 1, DARPA under grant No. FA8750-17-2- 0126, and DOE under awards No. DE-SC008923 and No. DE-SC014204.

## References

[1] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[2] Ian F. Akyildiz, Shuai Nie, Shih-Chun Lin, and Manoj Chandrasekaran. 2016. 5G roadmap: 10 key enabling technologies. *Computer Networks* 106 (2016), 17 – 48. https://doi.org/10.1016/j.comnet.2016.06.010

[3] Ian F. Akyildiz, Pu Wang, and Shih-Chun Lin. 2015. SoftAir: A software defined networking architecture for 5G wireless systems. *Computer Networks* 85 (2015), 1 – 18. https://doi.org/10.1016/j.comnet.2015.05.007

[4] Amazon.com Inc. 2018. Amazon EC2 Spot Instances. (2018). https://aws.amazon.com/ec2/spot/

[5] Apache Flink Community. 2014. Apache Flink. https://flink.apache.org/. (2014).

[6] Apache Storm Community. 2015. Apache Storm. https://storm.apache.org/. (2015).

[7] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of errors.* Ph.D. Dissertation. Royal Institute of Technology, Stockholm.

[8] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10).* ACM, New York, NY, USA, 89–108. https://doi.org/10.1145/1869459.1869469

[9] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. 2004. Load Management and High Availability in the Medusa Distributed Stream Processing System. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04).* ACM, New York, NY, USA, 929–930. https://doi.org/10.1145/1007568.1007701

[10] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. 2014. StreamJIT: A Commensal Compiler for High-Performance Stream Programming. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA).* Portland, OR. http://groups.csail.mit.edu/commit/papers/2014/bosboom-oopsla14-commensal.pdf

[11] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers (SIGGRAPH '04).* ACM, New York, NY, USA, 777–786. https://doi.org/10.1145/1186562.1015800

[12] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11).* ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

[13] Wei-Chiu Chuang, Bo Sang, Sunghwan Yoo, Rui Gu, Milind Kulkarni, and Charles Killian. 2013. Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications. In *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 21.

[14] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2.* USENIX Association, 273–286.

[15] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10).* USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855711.1855732

[16] Chenzhou Cui, Boliang He, Ce Yu, Jian Xiao, and Changhua Li. 2017. AstroCloud: A Distributed Cloud Computing and Application Platform for Astronomy. *arxiv.org* (2017).

[17] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (May 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[18] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16).* ACM, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2851141.2851148

[19] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. W. Lazio. 2009. The Square Kilometre Array. *Proc. IEEE* 97, 8 (Aug 2009), 1482–1496. https://doi.org/10.1109/JPROC.2009.2021005

[20] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).* ACM, New York, NY, USA, 299–313. https://doi.org/10.1145/2723372.2723726

[21] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11).* ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/1989323.1989356

[22] B. Farhang-Boroujeny. 2011. OFDM Versus Filter Bank Multicarrier. *IEEE Signal Processing Magazine* 28, 3 (May 2011), 92–112. https://doi.org/10.1109/MSP.2011.940267

[23] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[24] M.I. Gordon. 2010. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[25] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII).* ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1168857.1168877

[26] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A stream compiler for communication-exposed architectures. *SIGOPS Oper. Syst. Rev.* 36, 5 (Oct. 2002), 291–303. https://doi.org/10.1145/635508.605428

[27] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. 2011. Sponge: Portable Stream Programming on Graphics Engines. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI).* ACM, New York, NY, USA, 381–392. https://doi.org/10.1145/1950365.1950409

[28] G. Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Information Processing '74: Proceedings of the IFIP Congress,* J. L. Rosenfeld (Ed.). North-Holland, New York, NY, 471–475.

[29] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[30] K. T. Lam, Y. Luo, and C. L. Wang. 2010. Adaptive sampling-based profiling techniques for optimizing the distributed JVM runtime. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* 1–11. https://doi.org/10.1109/IPDPS.2010.5470461

[31] E.A. Lee and D.G. Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on* 100, 1 (1987), 24–35.

[32] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. STREAMSCOPE: Continuous Reliable Distributed Processing of Big Data Streams. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 439–453. http://dl.acm.org/citation.cfm?id=2930611.2930640

[33] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.* 22, 3 (July 2003), 896–907. https://doi.org/10.1145/882262.882362

[34] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. 2005. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC '05)*. USENIX Association, Berkeley, CA, USA, 25–25. http://dl.acm.org/citation.cfm?id=1247360.1247385

[35] P. J. Hall. Ed. 2005. *The Square Kilometre Array: An Engineering Perspective.* Springer.

[36] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 1–14.

[37] J Tseng R. Newman. 2011. Cloud Computing and the Square Kilometer Array *(Memo 134)*. The Square Kilometer Array, 1–21. http://www.skatelescope.org/uploaded/8762_134_Memo_Newman.pdf

[38] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.. In *NSDI*. 227–240.

[39] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. 2005. Cache Aware Optimization of Stream Programs. *SIGPLAN Not.* 40, 7 (June 2005), 115–126. https://doi.org/10.1145/1070891.1065927

[40] Sreekanth Setty. 2012. *VMware vSphereÂő 5.1 vMotion Architecture, Performance and Best Practices.* Technical Report EN-000986-00. VMware. 21 pages. http://www.vmware.com/resources/techresources/10305

[41] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin. 2003. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. 25–36. https://doi.org/10.1109/ICDE.2003.1260779

[42] Yosub Shin, Mainak Ghosh, and Indranil Gupta. 2015. Parqua: Online reconfigurations in virtual ring-based nosql systems. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*. IEEE, 220–223.

[43] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. 2011. Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications. In *2011 International Symposium on System on Chip (SoC)*. 14–21. https://doi.org/10.1109/ISSOC.2011.6089222

[44] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. 2007. StreamFlex: High-throughput Stream Programming in Java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 211–228. https://doi.org/10.1145/1297027.1297043

[45] The Square Kilometre Array. 2015. Seeing stars through the Cloud. http://skatelescope.org/news/ska-aws-astrocompute-cloud-computing-grant/. (2015).

[46] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[47] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 335–347. https://www.usenix.org/conference/atc17/technical-sessions/presentation/wei

[48] J. Wu, Z. Zhang, Y. Hong, and Y. Wen. 2015. Cloud radio access network (C-RAN): a primer. *IEEE Network* 29, 1 (Jan 2015), 35–41. https://doi.org/10.1109/MNET.2015.7018201

[49] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. 723–734. https://doi.org/10.1109/ICDE.2015.7113328

[50] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. 2005. Dynamic Load Distribution in the Borealis Stream Processor. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, Washington, DC, USA, 791–802. https://doi.org/10.1109/ICDE.2005.53

[51] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[52] W. Zhu, C. L. Wang, and F. Lau. 2002. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. 381–388. https://doi.org/10.1109/CLUSTR.2002.1137770