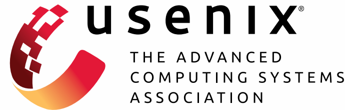

JetStream: Cluster-Scale Parallelization of Information Flow Queries

Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn, *University of Michigan*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/quinn

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).
November 2–4, 2016 • Savannah, GA, USA
ISBN 978-1-931971-33-1

Open access to the Proceedings of the
12th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

JetStream: Cluster-scale parallelization of information flow queries

Andrew Quinn, David Devecsery, Peter M. Chen and Jason Flinn
University of Michigan

Abstract

Dynamic information flow tracking (DIFT) is an important tool in many domains, such as security, debugging, forensics, provenance, configuration troubleshooting, and privacy tracking. However, the usability of DIFT is currently limited by its high overhead; complex information flow queries can take up to two orders of magnitude longer to execute than the original execution of the program. This precludes interactive uses in which users iteratively refine queries to narrow down bugs, leaks of private data, or performance anomalies.

JetStream applies cluster computing to parallelize and accelerate information flow queries over past executions. It uses deterministic record and replay to time slice executions into distinct contiguous chunks of execution called epochs, and it tracks information flow for each epoch on a separate core in the cluster. It structures the aggregation of information flow data from each epoch as a streaming computation. Epochs are arranged in a sequential chain from the beginning to the end of program execution; relationships to program inputs (sources) are streamed forward along the chain, and relationships to program outputs (sinks) are streamed backward. JetStream is the first system to parallelize DIFT across a cluster. Our results show that JetStream queries scale to at least 128 cores over a wide range of applications. JetStream accelerates DIFT queries to run 12–48 times faster than sequential queries; in most cases, queries run faster than the original execution of the program.

1 Introduction

Dynamic information flow tracking (DIFT) has emerged as an important tool for understanding and troubleshooting program behavior. Originally proposed by the security community [16], DIFT instruments an application binary to track data and/or control flow from global sources (e.g., program inputs) to global sinks (e.g., program outputs). Information flow analysis has proven to be helpful in a diverse set of domains that include forensic analysis [12], information provenance [6], privacy [7], application debugging [20], and troubleshooting of configurations [1, 2].

Unfortunately, dynamic information flow analysis can be painfully slow; depending on the granularity and amount of information tracked, execution slowdowns of up to one or two orders of magnitude are common. While this cost can be reduced by limiting analysis to managed languages such as Java or by restricting the types of queries that can be performed, *general-purpose* information flow analysis over binary code requires batch-style analysis for substantial programs. In other words, the user employing DIFT must run such analyses over the course of hours. DIFT would be much more powerful if analysis could be employed interactively; for instance, a user could refine a particular query by changing sources, sinks, the propagation function, the granularity of instrumentation, or the period of program execution over which the analysis is employed. The user could then narrow down the bug, misconfiguration, or privacy violation in a manner similar to traditional debugging techniques.

Our goal is to make DIFT queries interactive by parallelizing them across many cores in a compute cluster. With hundreds or thousands of cores, DIFT queries that previously took hours or days can complete in seconds or minutes, enabling refinement and iteration over multiple queries. Thus, a shared cluster can become a valuable resource for a large team of system operators or programmers who want to occasionally engage in an interactive debugging or troubleshooting session using DIFT tools. Usage scenarios for DIFT include both live analysis (as the program runs) and after-the-fact analysis (executed on a replay of an execution). We target the latter scenario.

Previous efforts at parallelizing DIFT have met with only limited success. Information flow is inherently difficult to scale (Ruwase et al. call it "embarrassingly sequential" [20]) because it tracks many fine-grained sequential dependencies between memory and register values. The set of dependencies at each step is a function of a large number of prior instructions executed by the program. Consequently, prior efforts have produced parallel versions that scale to only a few cores on a single machine, and no current approach can effectively leverage a commodity cluster to scale DIFT.

Our solution, JetStream, provides cluster-level scalability by computing information flow in two phases, each using a different form of parallelization.

The first phase is the **local DIFT phase**; it divides program execution into time segments (epochs) and assigns a separate core to compute the information flow

for each epoch. Each core determines the dependencies within its epoch that may be relevant to answering the overall query. It tracks sources and sinks that are identified explicitly in the query (e.g., network input and output); we call these *global sources/sinks*. It also tracks locations that may serve as links between global sources and sinks; we call these *local sources/sinks*. Local sources are all memory addresses and registers at the beginning of an epoch, and local sinks are all memory addresses and registers at the end of an epoch. The local DIFT phase parallelizes cleanly into separate partitions, with all dependencies between partitions resolved in the next phase.

Two challenges arise for the local DIFT phase. First, computing all dependencies that may be relevant to the query is too expensive. We address this challenge by deferring and avoiding work as much as possible. JetStream uses merge trees [20] to represent and manipulate dependency sets more efficiently. More importantly, it defers traversing these merge trees until the next (aggregation) phase; that phase avoids traversing the vast majority of tree nodes that do not lie on a dependency path between a global source and a global sink.

The second challenge is that each epoch must follow the same execution as the original execution, so that the aggregation of local DIFTs produces a result equivalent to a sequential DIFT. JetStream uses checkpointing and deterministic record/replay to divide an execution into epochs and perform the local DIFT for each epoch independently, yet consistently. JetStream uses lightweight statistics collected during the original execution of a program to partition the DIFT work equally, and it uses heavyweight statistics collected during the first query of an execution to better partition subsequent queries.

The second phase, called the **aggregation phase**, prunes and combines the information from the local DIFT phase to compute the final result, i.e., the relationship between global sources and global sinks across the entire execution. The cores in this phase are organized in a chain in order of program execution, and the computation is structured as a stream processing algorithm with pipeline-style parallelism. Each core resolves dependencies using information from one epoch's local DIFT phase, and the global query is answered via two streaming passes.

In the first streaming pass, the locations (registers and memory addresses) that are derived from global sources are passed *forward* along the chain (from the beginning to the end of execution). This information is used to prune deferred operations that do not depend on a global source. In the second streaming pass, the locations that propagate dependencies to a sink are passed *backward* along the chain. This lets JetStream prune deferred operations on which no sink depends.

The structuring of the aggregation is the most important factor in enabling JetStream to scale much better than prior approaches at parallelizing DIFT. Our insight is that a small amount of sequential information is necessary to avoid huge amounts of unnecessary work; this information is essentially the locations that depend on global sources (forward pass) and the locations on which global sinks will depend (backward pass). Streaming this data along a sequential chain allows most processing to occur in parallel, with the sequential limitation being essentially the time to pass a single data value from one end of the chain to the other; this is much less than the total query time even for hundreds of processors.

The contributions of this paper are:

- An algorithm for parallelizing DIFT that scales much better than prior approaches, enabling interactive (sub-minute) response times.
- Scalable and efficient support for tracking millions of distinct global sources and sinks at byte granularity, without restrictions on source-code availability, compute platform, or query type.
- A detailed evaluation of the remaining bottlenecks in accelerating DIFT through parallelization.

We have applied JetStream to run DIFT queries over seven desktop and server applications: Evince, Firefox, Ghostscript, Gzip, Mongodb, Nginx, and OpenOffice. Our results show that JetStream scales DIFT to at least 128 cores for these applications. It accelerates DIFT queries to run 12–48 times faster than sequential queries, and, in most cases, runs queries faster than the original execution of the program.

2 Motivation

DIFT is a fundamental analysis that is useful in diverse domains. For example, Arnold [6] uses DIFT for provenance queries that reveal how data values in files and application memory were derived. In forensics [12], DIFT has been used to answer questions such as: "How was my system compromised?" and "What data was leaked?" TaintDroid [7] and similar systems use DIFT to reveal whether an application execution leaks sensitive data. X-Ray [1] uses DIFT to identify misconfigurations that cause performance anomalies, and ConfAid [2] uses DIFT to identify misconfigurations that cause bugs. Poirot's [11] use of DIFT helps determine if a security vulnerability has been exploited.

Many of the above systems run complex DIFT queries on native binaries and can suffer from painfully slow DIFT query times. These systems are often forced to use batch-style computation, even though many would ideally be interactive in nature.

Consider a developer debugging an incorrect output value from a Web server. Using JetStream, she begins

by running a DIFT query that shows all program inputs from which the faulty value was derived. This alone is not enough to reveal the bug, so she runs an additional query tracking the inputs that led to a correct output. Comparing the results shows that inputs from a particular network connection led to the faulty output but not the correct one. Using this information, she discovers a bug in the code which parses network inputs. To see if this bug has impacted any file system state, the developer runs another query specifying all values from the faulty parsing code as global sources and all file system outputs as global sinks. She detects that no permanent state has been affected by her bug. Next she considers other forms of external output such as network messages.

Debugging the problem and determining the impact of the bug both require multiple DIFT queries. Further, phrasing the correct queries may be non-trivial and require multiple iterations to get helpful results. If each query takes hours to complete, then this process only makes sense for the most difficult bugs. In contrast, low-latency DIFT enables information flow analysis to be an integral part of the debugging process.

3 Background

We first describe two technologies on which JetStream builds: dynamic information flow tracking and deterministic record and replay.

3.1 DIFT

Dynamic information flow tracking, sometimes referred to as taint tracking, instruments applications to monitor data flow as programs execute. In its most general form, DIFT reveals which *global sources* causally affect which *global sinks* according to a *propagation function*. Global sources are typically external program inputs, such as bytes read from a file or a network socket, and global sinks are typically external outputs.

The propagation function specifies what information flows to track during program execution. For example, a basic data flow propagation function for the instruction $x = y + z$ would state that the sources on which x depends are the union of the sources on which y and z depend. Usually, DIFT tracks data flow (as we do in this work), but some DIFT systems also track implicit flows propagated via control flow.

When an application executes, DIFT assigns a *taint identifier* to each unique global source. For each location, it maintains a set of taint identifiers that shows the global sources on which that location currently depends, and it updates taint sets as instructions execute. At each global sink, DIFT outputs the set of taint identifiers of all locations written to the sink (e.g., the bytes sent to a network socket). Thus, DIFT produces a set of $\langle globalsource, globalsink \rangle$ tuples that describe how particular global sources and global sinks are related.

JetStream tracks global sources, global sinks, and dependencies at byte granularity using binary instrumentation inserted by Pin [14]. A single JetStream query may look for relationships between millions of distinct global sources and sinks. In contrast, many prior DIFT systems require source code or the use of a managed language runtime. Others track only whether any global source data propagates to a global sink and cannot determine *which* sources affect each sink—such systems cannot answer questions such as: "Which inputs affected this program value?" or "What data did I leak?"

A JetStream query contains a program execution to monitor, a filter that specifies the global sources, a filter that specifies sinks, and a propagation function. For instance, a provenance query [6] might wish to determine the lineage of the data in a particular file. The source filter would match all external program inputs and the output filter would match writes to a particular file. This would reveal which bytes in the file were derived from which sources. Alternatively, a privacy query [7] might specify reads from sensitive files as sources and network outputs as sinks. This would reveal what data was leaked over the network and how it was leaked. JetStream provides an interface for supporting custom propagation functions and supplies Arnold's copy, data, and index propagation functions [6] as defaults.

For complex applications, mapping all global sources to all global sinks at byte granularity produces far too much information (e.g., terabytes of data for some benchmarks in Section 5). Thus, filters are needed to extract the right information succinctly. This leads to refinement through iteration. Our goal is to make DIFT fast enough to be interactive, so that a user can issue multiple queries to search for the right information.

3.2 Deterministic replay

Deterministic replay allows the execution of a program to be recorded and reproduced faithfully. When a program first executes, all inputs from nondeterministic actions are logged; these values are supplied during subsequent replays in lieu of performing the nondeterministic operations again. Thus, the program starts in the same state, executes the same instructions on the same data values, and generates the same results.

JetStream derives several benefits from using deterministic replay. First, replay allows JetStream to partition a recorded execution into epochs and execute these epochs in parallel. Deterministic replay guarantees that the result of stitching together all epochs is equivalent to a sequential execution of the program. Second, replay allows an execution recorded on one machine to be replayed on a different machine. There are few external dependencies, since interactions with the operating sys-

	Instructions	Taint IDs	Merge Log	Live Set (forward pass)	Taint Tuples (backward pass)
Epoch 0	1. A = read() 2. B = read() 3. C = A + B	A: IN_0 B: IN_1 C: $M_0[0]$	$M_0[0] : \{IN_0, IN_1\}$	$\{A, B, C\}$	$\{< IN_0, OUT_0 >, < IN_1, OUT_0 >\}$
Epoch 1	4. D = X + Y 5. E = C 6. B = 0 7. Z = A[D]	D: $M_1[0]$ E: C_1 B: {} Z: $M_1[1]$	$M_1[0] : \{X_1, Y_1\}$ $M_1[1] : \{A_1, M_1[0]\}$	$\{A, C, E, Z\}$	$\{< OUT_0, C >\}$
Epoch 2	8. F = E 9. write(F)	F: E_2 OUT_0: E_2			$\{< OUT_0, E >\}$

Table 1: DIFT analysis of an example program.

tem and other external entities are nondeterministic and replayed from the log. Thus, the only requirement for replay is that the replaying computer has the same hardware architecture as the recording computer and that it runs a kernel modified to support replay. Finally, replay allows iterative queries over the same execution.

JetStream uses Arnold [6] to provide deterministic record and replay of multithreaded, multiprocess applications. Arnold's performance overhead is less than 10% for most workloads, and its storage overhead is reasonable even for continuous recording of a workstation.

4 Design and implementation

To parallelize a DIFT query, JetStream divides an execution into epochs and assigns each epoch to a different core. JetStream then evaluates the query in two phases: a *local DIFT* phase and an *aggregation* phase.

In the local DIFT phase, each core concurrently computes the relationships between sources and sinks within its epoch. A core can directly observe global sources and global sinks that occur during its epoch. However, some locations at the start of an epoch may depend on global sources from preceding epochs, and the local DIFT cannot know the actual dependencies because the local DIFTs for those preceding epochs are being executed concurrently. Thus, for all epochs but the first one, the local DIFT phase conservatively tracks all locations at the start of the epoch as *local sources* and assigns a unique *local source identifier* to each location at the epoch start. Similarly, the local DIFT cannot determine which locations at the end of an epoch will ultimately propagate to global sinks in succeeding epochs, so the local DIFT treats all locations at the end of the epoch as *local sinks*. A local DIFT phase thus tracks and reports dependencies between all sources (both local and global) and all sinks (both local and global).

In the aggregation phase, the cores organize as a chain in program execution order and communicate local DIFT results forward and backward along the chain to produce the final set of $\langle globalsource, globalsink \rangle$ tuples. We next describe these two phases in more detail.

Table 1 shows an example query in which JetStream finds all dependencies from global sources to global sinks in a simple program. Program execution is divided into three epochs (shown by the horizontal partitioning). The local DIFT phase is the region to the left of the double vertical bar, and the aggregation phase is the region to the right of the bar. Instructions 1 and 2 read data from global sources, and instruction 9 writes to a global sink.

4.1 Local DIFT

JetStream implements the local DIFT phase as a Pin tool. Executing an application with this tool attached is quite slow (e.g., 14–75x slowdown for the benchmarks in Section 5). There are two reasons: DIFT may add several additional instructions to track taint for each application instruction executed, and Pin dynamically adds the instrumentation to an application as it executes. The first is a fundamental cost of DIFT, while the second is a consequence of using a dynamic instrumentation tool.

The local DIFT phase for a given epoch first replays the application uninstrumented to advance its execution to the start of the epoch, a process we call *fast forwarding*. JetStream may start the replay from the beginning, or it may start from a checkpoint of application state taken during recording or during a previous query. Given the relative speed difference between instrumented and uninstrumented execution, starting from the beginning is reasonable for low numbers of epochs. As the number of epochs increases, fast forward time comes to dominate

total query time, and checkpoints are quite beneficial.

Next, JetStream attaches the DIFT tool to the application and Pin starts instrumenting the application to track dependencies. JetStream assigns a unique *source identifier* to each location modified by a global source in the epoch and to each location at the start of the epoch.

JetStream runs all threads of a multithreaded application on a single core to realize an important performance benefit: the instrumentation code does not need to obtain locks to synchronize access to the DIFT data structures because only one thread runs at any given time [21]. JetStream still fully utilizes the processor because each core runs a different epoch in parallel.

For each location, JetStream stores an integer *taint identifier* that represents the set of global and local sources on which that location currently depends. A taint identifier may be: (1) a global source identifier, (2) a local source identifier, or (3) an identifier that maps to a set of global and local sources. In the Taint IDs column of Table 1, IN_0 and IN_1 are global source identifiers, and C_1 and E_2 are local source identifiers that represent the taint of address C at the start of epoch 1 and the taint of address E at the start of epoch 2.

For each x86 instruction, the local DIFT tool reads the taint identifiers of the instruction's inputs and updates the taint identifiers of the instruction's outputs. Taint identifiers for registers are stored in a per-thread array, and taint identifiers for memory addresses are stored in a two-level page table. The tool decomposes the work for each instruction into a sequence of four sub-commands: `set`, `clear`, `copy`, and `merge`. The first three sub-commands are straightforward—`set` assigns a taint identifier to a location, `clear` assigns the NULL identifier to a location, and `copy` sets the destination's taint identifier equal to the source's. Thus, each of these sub-commands are low overhead integer operations. Table 1's Taint IDs column shows how the local DIFT tool updates the taint data structures: a `set` for instruction 1, a `clear` for instruction 6, and a `copy` for instruction 5.

The `merge` sub-command is used for instructions that combine dependencies, e.g., instructions 3, 4, and 7. For these instructions, the set of sources on which the output depends is the union of the sets of sources on which the inputs depend. Our original implementation tracked such sets explicitly, but this worked poorly. For some complex applications, the DIFT did not finish after running for hours, or the size of the sets exceeded the 256 GB memory of our server. Intuitively, the reason is that the set of tuples that relate all local sources to all local sinks can be as large as the size of the address space squared.

We therefore turned to an idea proposed by Ruwase et al. [20] in which sets of taint values are represented by a binary tree. Each `merge` operation generates a new taint identifier to represent the set union. JetStream writes an entry to a *merge log*, which contains the taint identifiers of the input to the `merge`. Thus, the merge log is a DAG sorted in temporal order, and each node (entry) in the merge log represents a binary tree of taint identifiers rooted at that node. Any merge node can be *resolved* to a set of source identifiers by performing a depth-first traversal of the tree rooted at that node.

In the example, instruction 4 creates a merge node $M_1[0]$ (each epoch has a distinct merge log, with the particular log denoted by the subscript). The node states that address D depends on whatever X and Y depend on at the start of epoch 1. Instruction 7 creates a merge node that has $M_1[0]$ as a child, so address X depends on whatever A, X, and Y depend on at the start of the epoch.

Using the merge log yields two benefits. First, it defers expensive set union operations until the aggregation phase; optimizations in that phase avoid the need to perform the vast majority of such unions. Second, the merge log uses much less memory than storing a set for each location. Memory usage is roughly proportional to the number of unique merge operations rather than the total size of all taint sets for every location. The cost of using a merge log is that JetStream must perform a tree traversal when it needs to resolve a root node to a set of source identifiers.

JetStream makes two enhancements to Ruwase et al's algorithm. First, it uses a hash table to cache recently-seen merge pairs and reuse merge nodes when duplicates are found. Second, whereas Ruwase et al. used the tree data structure only for abstract values (i.e., local source identifiers); JetStream also uses the tree structure for sets of global source identifiers, such as distinct bytes from different sources encountered during the local epoch (as for instruction 3 in the example).

At the end of an epoch, JetStream writes four datasets to a shared memory buffer: global source metadata, global sink metadata, the merge log, and the taint identifiers for all local sinks. The global source metadata describes each global source identifier (e.g., the system call that read the byte, the file the byte was read from, the offset within the file, etc.). Similarly, the global sink metadata describes each byte sent to a sink. Since application execution typically modifies only a small percentage of locations during a given epoch, the local sink identifiers for most locations will be the local source identifier of those locations. To save space, the local DIFT only outputs those local sink identifies where this relationship does not hold. These optimizations allow the output of the local DIFT phase to fit in the memory of modern servers (though it is still large, e.g., a few GB per epoch).

4.2 Partitioning

The time to produce an answer to a query depends on the longest local DIFT time for any epoch. Thus, to achieve good speedups, JetStream must partition local DIFT so that each core does roughly the same amount of work. To accomplish this, JetStream estimates the amount of time it will take to run local DIFT for any given interval of execution and defines epoch boundaries so that the estimated local DIFT time for each epoch is the same.

We estimate the local DIFT time for an interval of execution as a linear combination of three factors:

- **Fast Forward Time:** JetStream replays the application without instrumentation to advance execution to the start of the epoch. We estimate that this component of work is proportional to the user-level CPU time used for this portion of execution by the recorded application.

- **Instructions executed:** To track information flow for an interval of execution, JetStream must execute the instructions in that interval, as well as the instrumentation code that propagates dependencies among locations. This component of work is proportional to the number of instructions executed, which we estimate from the user-level CPU time used to execute the interval in the recorded execution.

- **Unique instructions executed:** Pin instruments an instruction when it is executed for the first time. With Pin, instrumentation cost is a significant portion of the overall DIFT time, especially for short intervals in which each instruction may only be executed a few times. As JetStream parallelizes the DIFT work across more cores, each interval becomes shorter, and the relative cost of instrumenting instructions increases. This component of work is proportional to the number of *unique* instructions executed. During recording, we read processor performance counters via the `perf_events` API to estimate the number of unique instructions executed by sampling the instruction pointer (we sample every 32 L1 instruction cache read misses for user-level code). When executing the first query for an execution, we use dynamic instrumentation to measure the actual number of unique instructions executed during an interval; this adds little overhead compared to DIFT instrumentation.

To avoid confounding testing and training in our evaluation, we choose the constants in the model for the first query of an execution by running a linear regression over data from the *other* benchmarks in our set. The coefficient of determination (R^2 value) for these regressions is 0.86–0.87. Due to the high overhead of instrumenting code with Pin, the cost of inserting instrumentation (proportional to unique instructions executed) usually dominates the cost of running the instrumented code (proportional to instructions executed), especially for small epochs.

For subsequent queries of a given execution, we run a linear regression over the performance data gathered during the first query. This produces a much better R^2 value of 0.985. We also add the number of merges that occurred during each interval to our model, and that change slightly increases the R^2 value to 0.989.

JetStream partitions the recorded execution into n epochs of roughly equal local DIFT time as estimated by the above model, where n is the number of available cores to run the query. This process is conceptually simple, but a complication is that the total local DIFT time depends on the particular partitioning chosen because an instruction that is executed in multiple epochs will incur an instrumentation cost in each of those epochs. We solve this problem by using a hill-climbing algorithm in which each iteration updates the estimate of the total local DIFT time for the query, and the new estimate is used to calculate a better partitioning in the next iteration. Usually, this process converges after a small number of iterations.

4.3 Aggregation

The aggregation phase produces the set of $\langle globalsource, globalsink \rangle$ tuples that are related by the propagation function. Within a single epoch, a global source and global sink are related if the global sink either has the global source's identifier or if it has the identifier of a merge node and that node resolves to a set that contains the global source's identifier. If the global source and sink are in adjacent epochs, then they are related if there exists a location L at the boundary of the two epochs such that, in the first epoch, the local sink identifier of L depends on the global source, and in the second epoch, the global sink depends on the local source identifier of L. If one or more epochs separate the epochs of the global source and sink, then there must be multiple such relationships forming a continuous path from source to sink.

In Table 1, such a path exists between the global sources of instruction 1 and 2 and the global sink of instruction 9. In epoch 0, resolving the merge tree for address C reveals that it depends on both global source 0 and global source 1. In epoch 1, the final value of E depends on the value of C at the beginning of the epoch. In epoch 2, instruction 9 writes address F, which depends on location E at the beginning of the epoch. Determin-

ing the complete relationship between global sources and global sinks requires aggregating the data from the local DIFT phase of each epoch.

4.3.1 Parallelizing aggregation: A failed attempt

To meet our performance goals, both the local DIFT and aggregation phases must scale well with the number of cores. Our first approach to constructing a parallel aggregation phase was based on a tree-like merge of local DIFT information. First, each individual epoch produces a map of all $\langle source, sink \rangle$ tuples where sources and sinks may be either local or global. For each sink with a taint identifier that represents a merge node, the map is generated by a depth-first traversal of the tree rooted at that node to resolve the set of source identifiers. This step is performed in parallel for all epochs, and caching is used to avoid revisiting tree nodes. If both the source and sink in a tuple are global, then the tuple is immediately output and removed from the map. Such tuples represent dependencies that can be computed solely on the DIFT information in a local epoch.

Next, we merge maps for pairs of adjacent epochs. For all locations, L, if there exists a tuple $\langle source, L \rangle$ in the first epoch and a tuple $\langle L, sink \rangle$ in the second epoch, then this step adds the tuple $\langle source, sink \rangle$ to its map. Since two epochs are involved, this step is parallelized across two cores. As before, if the sources and sinks in a tuple are both global, the tuple is immediately output and removed from the map. Merges are performed in a binary tree, merging sets of 4, 8, 16, etc. epochs, using the same approach as above. The number of cores participating in each merge grows proportionally, and the number of merge steps is logarithmic in the number of epochs.

Unfortunately, this algorithm performed very poorly. Traversing the merge log for each end value and generating sets of start values was extremely time-consuming, even with caching and reuse of intermediate results. Even worse, for most of our applications, some of the merged maps failed to fit in 256 GB of memory. Our analysis showed that the reason for this behavior was that we were doing far more work than we needed to: the vast majority of merge nodes visited and values in the merged maps were not actually on a path between a global source and a global sink. However, because no epoch knew the full set of global sources and sinks when creating or merging maps, each had to calculate all dependencies that could possibly be used.

We concluded from this failed attempt that a fully-parallel aggregation phase is infeasible because it vastly increases the total work done. To fix this, aggregation must use data about global sources and sinks to generate less intermediate data and to traverse fewer merge nodes.

4.3.2 Backward pass

Our next approach to aggregation was to structure the computation as a stream processing algorithm that scales via pipeline-style parallelism. We arrange the epochs in an ordered chain. In parallel, each epoch processes any global sinks encountered during the epoch. The JetStream aggregator checks the taint identifier for each byte sent to a global sink. If the taint identifier is a global source (i.e., if the global source and sink are in the same epoch), the aggregator immediately outputs a $\langle globalsource, globalsink \rangle$ tuple. If the taint identifier is a local source identifier L, the aggregator sends a $\langle L, globalsink \rangle$ tuple to the *previous* epoch in the chain. Epochs on the same machine communicate via a shared memory buffer; epochs on different machines communicate via a TCP socket. If the taint identifier is a merge log node, the aggregator resolves the set with a depth-first traversal of the tree rooted at that node. For each unique source identifier in the set, it either outputs a $\langle globalsource, globalsink \rangle$ tuple or sends a $\langle L, globalsink \rangle$ tuple to the previous epoch.

When the aggregation phase for an epoch receives a $\langle L, globalsink \rangle$ tuple from the succeeding epoch, it checks the epoch's local sink taint identifier for L. This is either a local source identifier, a global source identifier, or a merge node identifier that resolves to a set of source identifiers. For each global source, the aggregator outputs a $\langle globalsource, globalsink \rangle$ tuple, and for each local source L', it sends a $\langle L', globalsink \rangle$ tuple to the preceding epoch.

The last epoch sends a sentinel value to its preceding epoch after it has finished processing its sinks; when an epoch reads the sentinel, its work is done as no more tuples will be forthcoming. It then sends the sentinel to its predecessor.

The last column of Table 1 shows the backward pass. Epoch 2 determines that OUT_0 depends on location E and passes that tuple to epoch 1. Epoch 1 determines that E depends on C, so passes the tuple $\langle OUT_0, C \rangle$ to epoch 0. Epoch 0 resolves the merge tree rooted at $M_0[0]$ and outputs tuples relating OUT_0 with both IN_0 and IN_1.

The major advantage of this streaming algorithm is that no epoch will process a merge node or send a tuple to a proceeding epoch unless the node/tuple represents a location that propagates to some global sink according to the propagation function. In the example, no merge nodes in epoch 1 are visited. This vastly reduces the amount of aggregation work. The potential disadvantage of this algorithm is that we have added a sequential step; each tuple must flow from global sink to global source, passing through all intermediate epochs. Our results show that this has only a minor effect on overall query time since each core can still process tuples in parallel. In other words, the latency of passing a tuple

through all epochs in the chain is very small compared to the query time, just as the sequential time to execute a machine instruction in a pipelined CPU is trivial compared to the time spent operating with a full pipeline.

Our results show that this algorithm, which we will refer to as the *backward pass* produces reasonable aggregation costs for some simple applications/queries, but it still takes too long for complex applications/queries. The reason is that we are still visiting too many merge nodes and creating too many tuples that are not ultimately on the path between a source and a sink. Thus, we found it necessary to also add a streaming *forward pass* that propagates information about which locations are related to global sources along the chain of epochs.

4.3.3 Forward pass

The forward pass runs prior to the backward pass. For each epoch, the forward pass first calculates a *reverse index* that has the same vertexes as the merge log DAG but that has edges in the opposite direction. The reverse index is also a DAG; depth-first traversal from a given local or global source yields the set of sinks that depend on that source. Each epoch builds its reverse index by first visiting all merge log nodes in temporal order, then visiting all local sinks. This step is fully parallelized since the reverse index can be computed purely with local information for each epoch.

Next, for each byte read from a global source, the aggregator does a depth-first traversal of the reverse index to determine the set of local sinks that depend on any source. It passes these sink locations to the succeeding epoch in the chain (forward in time). Here, the aggregator is only determining that a given local sink is tainted by any global source; it is not identifying a particular global source that has tainted the local sink. Therefore, the aggregator passes a local sink to the succeeding epoch at most once, and it visits each node in the reverse index at most once. It sets a `visited` bit for each local sink and merge node to avoid duplicate work.

As the aggregator receives locations from the prior epoch, it does a depth-first traversal of the reverse index to determine which (if any) additional local sinks depend on that location. It sends the locations associated with those local sinks to the succeeding epoch. The aggregator also retains the complete set of locations obtained from the prior epoch; this *live set* is the set of all local sources that depend on any global source.

Similar to the backward pass, the first epoch sends a sentinel token as soon as it finishes processing global sources. Once an epoch receives the sentinel, its live set is complete; the epoch then sends the sentinel to its successor.

In Table 1, the Live Set column shows the forward pass. At the end of the first epoch, locations A, B, and C depend on at least one global source. The second epoch adds Z to this set because it depends on A and adds E to this set because it depends on C. Additionally, the second epoch removes B from this set because its taint value was cleared.

Once an epoch knows its live set, it prunes its merge log. The aggregator processes merge log nodes sequentially. Any local source not in the live set for that epoch cannot depend on a global source. So, if a child of a merge node is a local source identifier, and the local source is not in the live set, the child is replaced by a NULL identifier. If a merge node has two NULL children, no members of its source set depend on a global source. Any identifier in the merge log that refers to such a node is also replaced with a NULL value. Essentially, this is a garbage collection in which any node known to be unrelated to a global source is removed. This garbage collection can substantially prune the merge log. Each epoch can run the prune in parallel once it knows its live set. Thus, the only sequential component of the forward pass is the propagation of live set values. In Table 1, epoch 1 prunes merge node $M_1[0]$ because it does not depend on any global source. $M_1[1]$ is updated to $\langle A_1, NULL \rangle$.

By inserting a forward pass, JetStream guarantees that all merge nodes processed and all tuples generated during the backward pass are on a path between a global source and global sink. This vastly reduces the number of nodes processed and tuples generated, making the backward pass more efficient. Note that although the forward pass itself must visit all nodes tainted by a global source (even those that do not lead to a global sink), the forward pass does much less work than the backward pass because it tracks only whether or not a location depends on a global source. It does not identify the specific source(s) on which the location depends.

4.3.4 Pre-pruning

JetStream uses one final optimization to improve aggregation performance. During an epoch, many values in memory or registers are overwritten before the epoch ends. If a merge log node does not propagate to either a local or global sink, then it can be removed from the log based solely on information available from that epoch. We call this step *pre-pruning*. JetStream does pre-pruning via a mark-and-sweep garbage collection over the merge log. It iterates through all sinks; if a sink has the taint identifier of a merge log node, JetStream marks the merge node as referenced. Then, JetStream iterates backward through the merge log. For each child in a merge log entry that refers to a prior merge log node, JetStream marks the prior merge log node as referenced. It discards all unmarked nodes and compacts the merge log. This reduces the number of merge log nodes that need to be processed later during both the forward and backward passes.

Benchmark	Replay Log Size (MB)	Replay Time (seconds)	Sequential DIFT Time (seconds)	Global Sources	Global Sinks	Dependencies
Gzip	0.03	2.98	109.23	64352941	48791393	36586765
Ghostscript	0.12	1.03	76.90	2514067	176009	14682254
Evince	2.90	13.47	234.30	10302852	104061604	346305
Nginx	30.65	4.75	196.51	10412627	35000000	5000000
Mongodb	37.02	22.79	309.99	8863855	116592809	76042962
OpenOffice	15.25	7.55	418.03	9946659	32110959	14599069
Firefox	24.80	67.42	1838.70	920029	1636119	131476

Table 2: Benchmarks used in the evaluation.

4.3.5 Summary

For each epoch, JetStream performs the following operations: (1) It runs the program without instrumentation from the start or from the nearest checkpoint to the beginning of the epoch. (2) It attaches a Pin tool and performs a local DIFT until the end of the epoch. (3) It pre-prunes the resulting local DIFT output to eliminate merge log nodes that cannot lead to a global sink. (4) It performs a forward aggregation pass to further prune the merge log by excluding any node that does not depend on a global source. (5) It performs a backward aggregation pass to generate $\langle globalsource, globalsink \rangle$ tuples; only merge nodes and locations on the path between a source and a sink are visited during this pass. Almost all of these steps can be performed in parallel for each epoch. The exceptions are the propagation of source dependencies in the forward path and $\langle location, sink \rangle$ tuples in the backward pass. These sequential steps are structured as stream processing along the epoch chain to maximize the work done in parallel.

5 Evaluation

Our evaluation answers the following questions:

- How well does JetStream scale DIFT?
- What are the remaining scalability bottlenecks?
- What is the impact of query optimizations?

5.1 Experimental Setup

JetStream uses the Arnold record and replay system [6] and the Pin dynamic instrumentation framework [14]. We evaluated JetStream using a CloudLab [19] cluster of 32 r320 machines (8-core Xeon E5-2450 2.1 GHz processors, 16 GB RAM, 10 Gb NIC). We envision running JetStream on an even larger cluster, but we could only reliably get a 32 machine cluster from CloudLab. Since these machines have a relatively small amount of RAM (16 GB) and DIFT queries are memory-intensive, we use only 4 cores per machine, leaving the experimental setup with 128 effective cores. For all experiments, we report the mean of 5 trials and show 95% confidence intervals.

5.2 Benchmarks

We evaluate JetStream with seven benchmarks chosen to represent common desktop and server workloads:

- **Gzip** – Zip a large file.
- **Ghostscript** – Convert a research poster from PostScript to PDF.
- **Evince** – Open and view a research paper.
- **Mongodb** – Yahoo cloud server benchmark [5].
- **Nginx** – Serve static content.
- **OpenOffice** – Edit a conference presentation.
- **Firefox** – A long Facebook browsing session.

For Gzip, Ghostscript, Evince, Mongodb, and Nginx, the query asks for dependencies between all command line, network, and file system inputs and all such outputs. Running the all-to-all query for OpenOffice and Firefox generated over 1 TB of data before we stopped the query. Thus, the OpenOffice query only considers file system data from the user's home directory to be sources, and the Firefox query considers cookie data to be sources and network output to specific sites (about 10% of total output) to be sinks. We use Arnold's data flow propagation function for all queries.

Table 2 shows a summary of the benchmarks. These are complex queries: most consider millions of distinct source and sinks, and most generate millions of dependencies. We show the time to replay each benchmark without instrumentation and the sequential DIFT time on a replay as baselines. We do not show the time for the original benchmark to run because that time depends on user think-time (for interactive applications), network delays, idle time (for server applications) and external output. When the benchmark is not CPU bound, DIFT overheads can be underestimated. Replay time, against which we compare, can already be one or two orders of magnitude faster than the original execution time [6]. We also report the compressed replay log size for each benchmark.

5.3 Scalability

We first evaluate the scalability of JetStream queries. Figure 1 shows the speedup of executing the first query for each benchmark on a log-log scale as we vary the number of cores from 1 to 128. Results are normalized to evaluating a query using a sequential algorithm on a single core; the black diagonal line shows ideal speedup, and a horizontal line would show no speedup. Overall, JetStream accelerates DIFT queries by 8–28x with

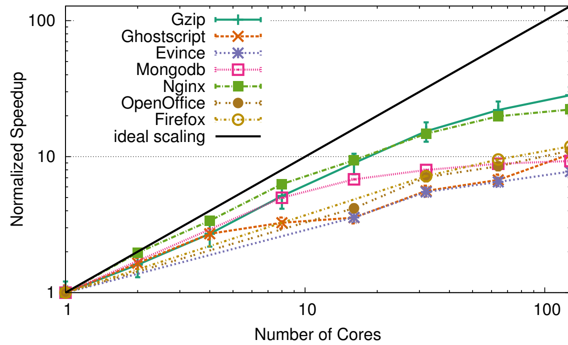

Figure 1: First Query Scalability - JetStream's scalability from 1 to 128 cores

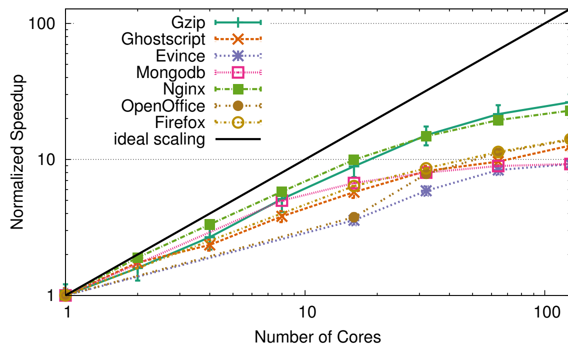

Figure 2: Scalability After Repartitioning - JetStream's scalability after incorporating its improved partitioning model

a mean of 13x using 128 cores. All benchmarks continue to scale through 128 cores, but some (e.g., Evince) scale less well at high numbers of cores.

We find that the biggest bottlenecks to scalability of the first query are (1) the epoch partitioning is often imbalanced, resulting in delays due to tail latency, and (2) the fast forward time becomes a bottleneck as query time approaches the replay time (since we need to replay the application from the beginning to start an epoch).

We address the first bottleneck by gathering data about unique instructions executed during the first query and improving the partitioning. Figure 2 shows the impact of repartitioning for the second query. With this optimization, JetStream scales the DIFT queries by 9–26x, with a mean of 14x. Repartitioning improves performance for all benchmarks except Gzip; in the case of Gzip, the model generated with less-detailed statistics is actually a better predictor of performance than the model generated with more-detailed statistics.

We address the second bottleneck by taking intermediate checkpoints during the first query. Figure 3 shows the scalability of the second query when using both repartitioning and checkpointing. JetStream scales the DIFT queries by 12–48x, with a mean of 21x. All benchmarks continue to scale up to 128 cores, though the pace of scaling diminishes with larger number of epochs. At 128 cores, the Gzip and Mongodb queries execute

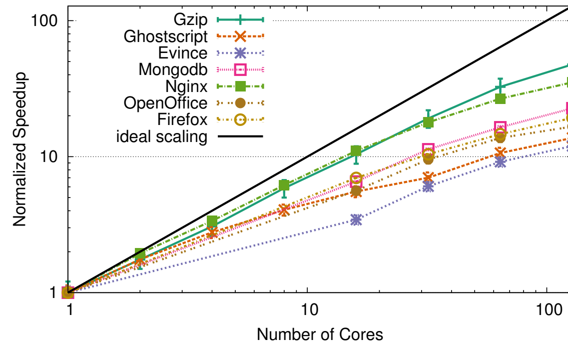

Figure 3: Second Query Scalability - JetStream's scalability after improving partitioning and checkpointing

faster than their sequential replay times, and all benchmarks except Ghostscript execute faster than the original execution time of the application.

5.4 Analysis of first-query bottlenecks

Next, we examine results for individual benchmarks in more detail and identify scalability bottlenecks. Figures 4 and 5 show stacked bar graphs for each benchmark at 128 cores; results for the first query are in the left column, and results for the second query are in the right column.

Each stacked bar in a graph shows the time spent in different query stages for a single epoch; the epochs are ordered left to right by the order of the time slices in the application execution. The bottom region, labeled fast forward, shows the time for application execution to reach the start of the epoch. Instrumentation is the time required for Pin to instrument instructions, and analysis is the time to execute that instrumentation. The split between these two values is estimated by assuming that the instrumentation cost is equal to the unique instructions executed term from the model in Section 4.2 (which has an R^2 value of 0.989) as we cannot directly distinguish these two values.

Pre-prune, forward pass, prune, and backward pass show the time spent in each aggregation stage. The sequential constraints of the forward pass and backward pass are shown by the gently sloping lines at the top of each region: one epoch's forward or backward pass cannot complete until the prior epoch in that pass has completed. The total time to complete the query is given by the height of the first stacked bar; the first epoch is the last to complete aggregation because of the sequential nature of the backward pass.

Outlier epochs caused by the result of poor partitioning can be detected by variance in the tops of the analysis regions (the combination of the fast forward, instrumentation, and analysis phases). All of our benchmarks except Gzip and Mongodb noticeably benefit from improved partitioning. For example, comparing the first and second queries of OpenOffice (Figures 5a and 5b)

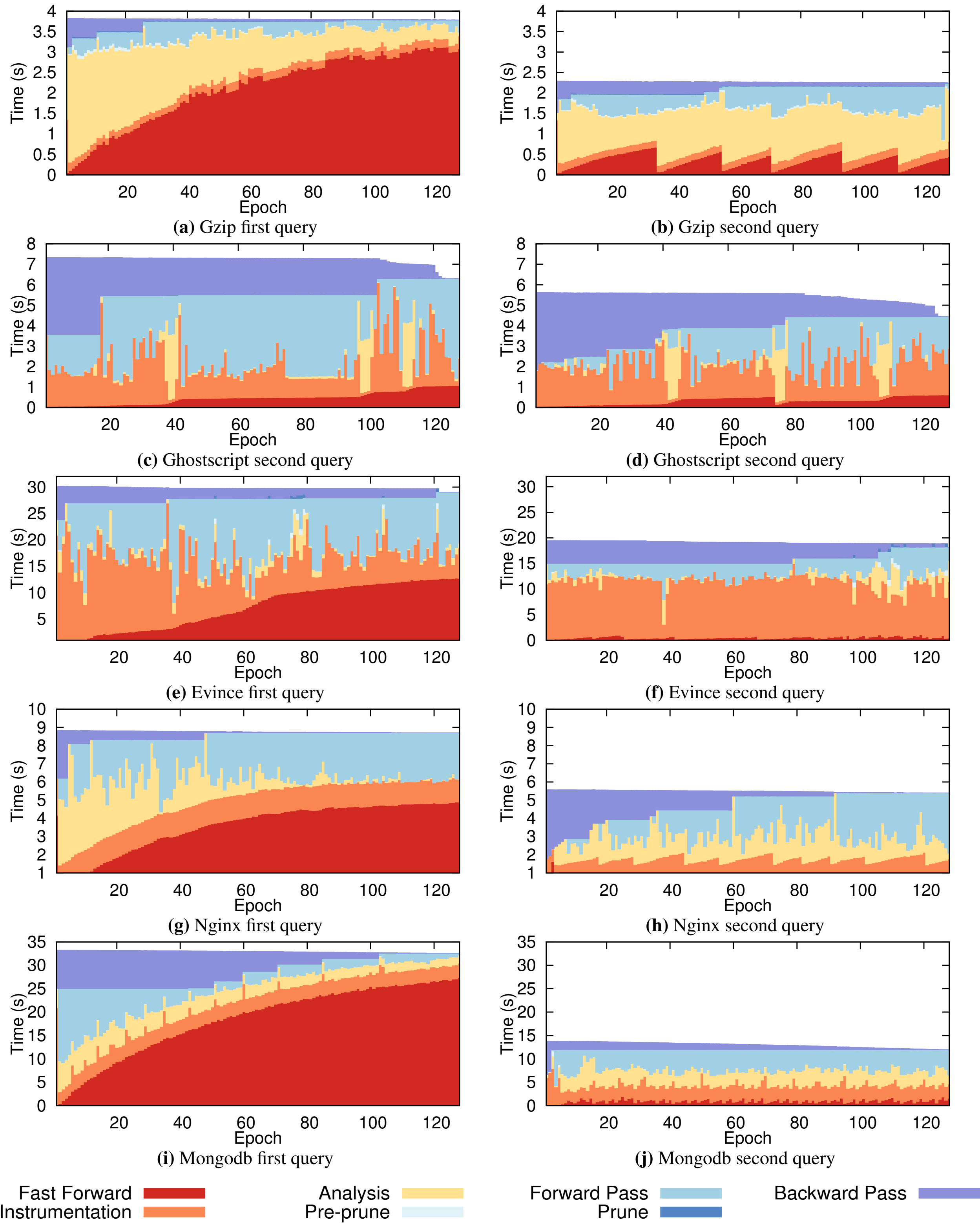

Figure 4: Breakdown of query processing time for 128 cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.

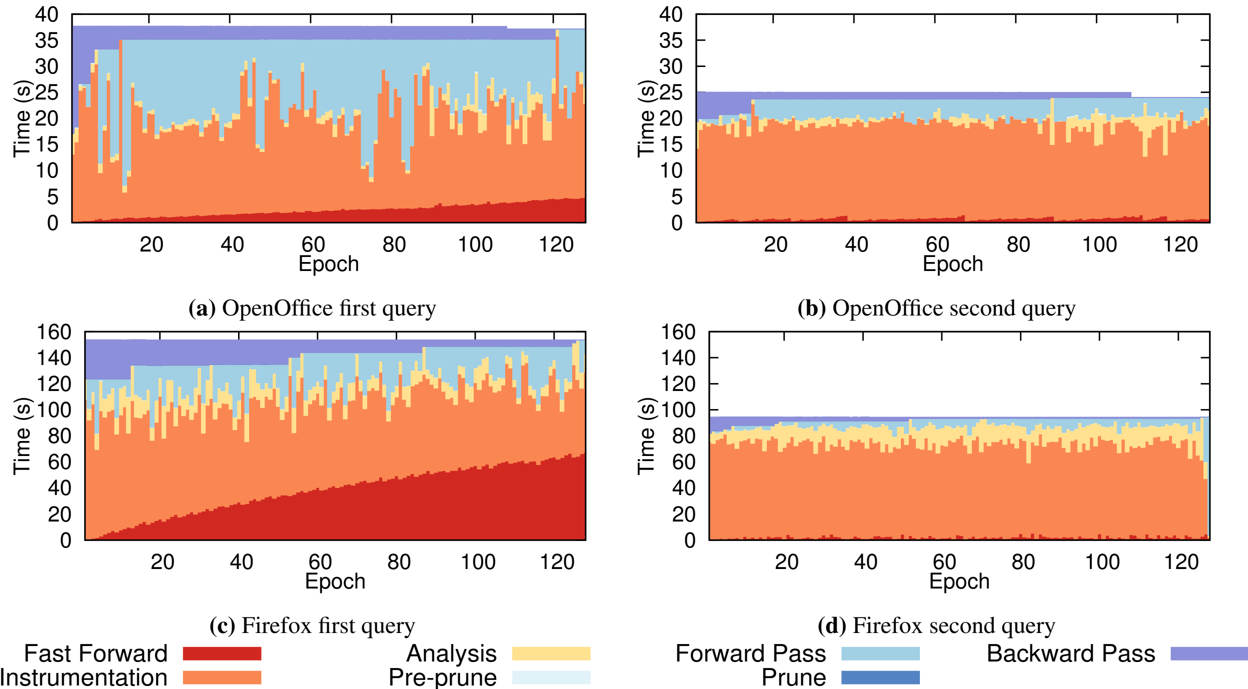

Figure 5: Breakdown of query processing time for 128 cores. Each stacked bar shows one core processing an epoch. From bottom to top, the shaded regions within each bar show the time spent fast forwarding, doing dynamic instrumentation, running DIFT analysis, pre-pruning, performing the forward pass, pruning the merge log and performing the backward pass.

shows the benefit of improving the partitioning between the first and second queries. Reducing outliers leads to substantially faster second query times for these benchmarks.

The effects of checkpointing in JetStream can be seen in all of our benchmarks. For example, comparing the first and second queries of Gzip (Figures 4a and 4b) shows the dramatic effect that checkpointing can have on query latency. The primary reason that this benchmark does not scale well for the first query is that the query time approaches the replay time of the benchmark—this is shown by fast forward being a large component of the last epoch time for the first query. In contrast, the fast forward times in the second query are much smaller.

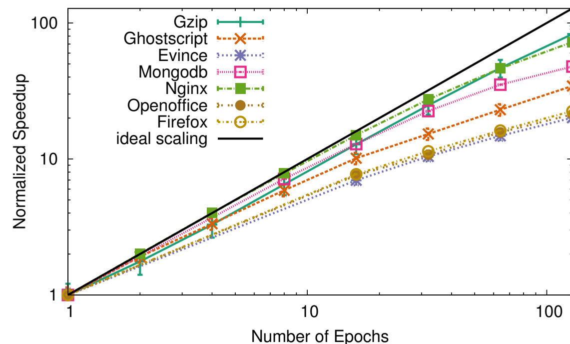

Figure 6: DIFT Scaling - Scalability of local DIFT (excluding fast-forward time) for different numbers of cores normalized to local DIFT (excluding fast-forward time) for one core.

5.5 Analysis of second-query bottlenecks

We next look at second query performance and bottlenecks. Interestingly, the specific bottlenecks vary from benchmark to benchmark.

For Evince (Figure 4f), OpenOffice (Figure 5b), and Firefox (Figure 5d), Pin instrumentation time dominates total query time. Pin instrumentation time also impacts Ghostscript (Figure 4d) to a lesser degree. To explore this issue in more detail, Figure 6 shows the speedup for just instrumentation and analysis. All benchmarks scale up to 128 cores, but not ideally.

There are two main factors that limit instrumentation and analysis scalability: (1) JetStream must taint all local sources at each epoch boundary, and (2) Pin instruments an instruction in every epoch in which that instruction occurs, so dividing the program into smaller epochs increases the total instructions instrumented. We isolated the cost of (1) by running a sequential query on one core that retaints each address at epoch boundaries. This does the exact same work as the parallel version, and it produces the same results; however, Pin instruments each instruction only once across all epochs. As Figure 7 shows, the overhead added by tainting local sources is relatively small (3–25% of the sequential DIFT query). When this overhead is parallelized over 128 cores, it should have little effect on query time. Additionally, our model from Section 4.2 shows that unique instructions correlate very

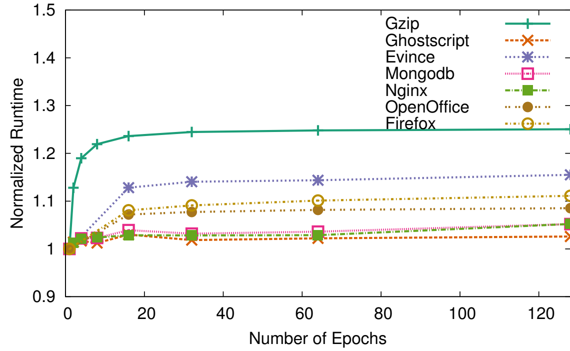

Figure 7: Retainting Overhead - Overhead of tainting local sources at the beginning of each epoch.

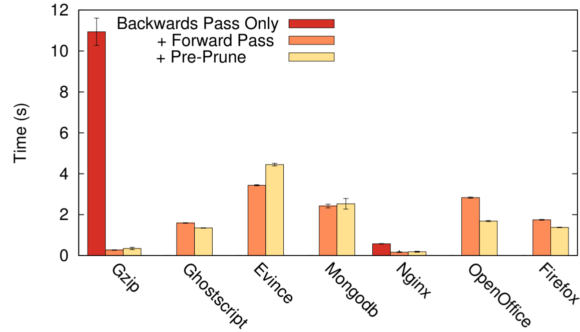

Figure 8: Optimization Effectiveness - Effect of aggregation optimizations at 128 cores. Experiments that did not complete are left blank.

highly with instrumentation and analysis time.

Switching from a dynamic to static instrumentation tool, using techniques that reduce the amount of dynamic instrumentation [8], or employing a low-overhead binary instrumentation platform (e.g., Protean code [13]) could reduce instrumentation time. Alternatively, we could take checkpoints that include already-instrumented code, as is done by Speck [17].

Poor partitioning is a significant component of overall query time for Ghostscript (Figure 4d), Nginx (Figure 4h), and Mongodb (Figure 4j). There are two separate reasons for poor partitioning in these benchmarks.

JetStream gathers statistics about query execution after each system call is executed. Ghostscript contains long regions of computation without a system call. At 128 epochs, JetStream must split some of these regions into multiple epochs. It divides these regions crudely based on the number of entries in the replay log; this crude metric often mispredicts the actual query execution time for the split epochs. Gathering statistics at finer-grained intervals would reduce outliers.

Outlier epochs in Nginx and Mongodb occur due to variance in the amount of taint processing done in each epoch. Outliers are correlated with large numbers of tainted sources and/or sinks in the epoch. Currently, our partitioning tool cannot determine which sources and sinks will be tainted in advance, but JetStream could potentially gather more statistics about sources and sinks during the first query, which could help the partitioning tool make such a determination for subsequent queries.

Aggregation plays a minor role in query time for most benchmarks. The speed of the forward pass is seen in the slope of the top of this region across epochs. Similarly, the speed of the backward pass is given by the slope of the top of that region. The total area for these two regions is less relevant since the sequential constraints mean that epochs will sometimes be idle waiting for data to arrive from predecessor epochs. We see negligible forward pass time across all benchmarks. Backward pass time is most noticeable for Ghostscript (Figure 4d), Mongodb (Figure 4j) and OpenOffice (Figure 5b), as shown by the noticeable slope of the top region in each graph. Better caching heuristics may improve the backward pass for these benchmarks.

5.6 Optimizations

We next evaluate the costs and benefits of optimizations employed by JetStream. We first measure the benefit of two aggregation optimizations: the forward pass and pre-pruning. To isolate aggregation cost from outliers in the local DIFT stage, we let all epochs finish local DIFT before beginning aggregation. This is the worst case for aggregation costs since individual epochs cannot pre-prune or construct the reverse index while waiting for prior epochs to finish local DIFT.

Figure 8 shows the isolated cost of aggregation for 128 cores. If aggregation performs only the backward pass (omitting the forward pass and pre-pruning), then only Gzip and Nginx complete; aggregation runs out of memory on all other benchmarks. For Gzip and Nginx, adding the forward pass improves isolated aggregation time by 98% and 71%, respectively.

The pre-prune optimization appears less effective. It decreases isolated aggregation time for Firefox, OpenOffice, and Ghostscript, but increases it slightly for Mongodb, Evince, and Gzip. We conclude that JetStream's policy of always pre-pruning is likely suboptimal; an adaptive policy that only pre-prunes when spare CPU cycles are available would be better.

We also measured the extra costs to optimize partitioning. We measured the time to profile L1 instruction cache misses for CPU-intensive benchmarks (Gzip and Ghostscript); the average overhead was 3.1%. The average overhead imposed by taking checkpoints during the first query was only 0.7% since each epoch takes at most one checkpoint. Finally, the average overhead of tracing unique instructions during the first query was 1.5%.

6 Related work

JetStream is the first system to parallelize DIFT across a cluster, and it is the first system to efficiently track millions of global sources, global sinks, and dependencies. Several prior systems have parallelized DIFT across the cores of a single machine. To achieve cluster-level scalability, JetStream's main contribution is parallelizing the aggregation of local DIFT data while minimizing the communication between cores.

Like JetStream, Speck [17] partitions an execution into epochs and performs local DIFT for each epoch. Speck tracks only a single label (tainted or untainted). Speck's local DIFT produces a log of sub-commands, which it then optimizes to achieve an up to 6x reduction in log size. Aggregation is done sequentially over the optimized log. This limits the speedup achieved by Speck to only 2x on a 8-core machine.

Ruwase et al. [20] partition an execution into epochs and perform local DIFT on each core using custom hardware [4]. JetStream's merge log optimization is derived from this work; thus, the local DIFT phases of the two systems are similar. However, Ruwase et al. perform aggregation sequentially, and this limits scalability. Like Speck, their system tracks only a single label. TaintPipe [15] partitions DIFT into epochs and tracks taint as symbolic formulas inside each epoch. TaintPipe also performs aggregation sequentially. It is unclear how symbolic tracking can scale efficiently to millions of labels and dependencies.

JetStream focuses on after-the-fact analysis, while prior DIFT parallelization has focused on live analysis during execution. Live analysis runs only a single predefined query, but it is suitable for security use cases in which sensitive actions such as sending network output need to be blocked based on the DIFT results (Speck and Ruwase et al. delay output to support this functionality, while TaintPipe does not). In contrast, after-the-fact analysis is suitable for tasks like forensics [12], debugging [20], configuration troubleshooting [1, 2], analysis of privacy leaks [7], and provenance [6]. No prior system has parallelized after-the-fact DIFT.

Many systems have explored how to make DIFT itself faster. One promising idea is decoupled execution, in which the DIFT work is split into an instrumentation thread and an analysis thread. ShadowReplica [8] combines decoupled execution with static analysis to reduce the amount of instrumentation that Pin must perform. TaintPipe combines decoupled execution with another form of static analysis: taint abstractions for commonly used function. libdft [10] provides several low-level optimizations for accelerating Pin-based DIFT. Profiling and/or static analysis can also reduce the cost of dynamic instrumentation [3, 9, 18].

These ideas are orthogonal to the speedups that JetStream provides through parallelization. In fact, our evaluation shows that Pin dynamic instrumentation is often the scalability bottleneck after JetStream parallelization, so incorporating these optimizations into JetStream is a very promising direction for future work.

7 Conclusion

JetStream enables interactive DIFT over past executions by parallelizing queries across a cluster. It uses deterministic record and replay to divide an execution into epochs and execute a local DIFT for each epoch on a separate core. It aggregates results from local DIFTs by arranging epochs in a sequential chain according to the order of program execution and using a pipeline-like stream processing algorithm to pass information about global sources and sinks along the chain. For future work, we plan to explore novel debugging and forensics applications enabled by JetStream.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Shan Lu, for their thoughtful comments. We also thank Mike Chow and Xianzheng Dou for their help understanding the Arnold code base. We thank the CloudLab team which helped us deploy our system on their experimental platform. This work has been supported by the National Science Foundation under grants CNS-1513718 and CNS-1421441. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[2] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[3] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, October 2008.

[4] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Mchiael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[6] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[7] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, October 2010.

[8] Kangkook Jee, Vasileios P. Kermerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[9] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kermerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromyrtis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Network and Distributed System Security Symposium*, San Diego, CA, February 2012.

[10] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, 2012.

[11] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation*, Hollywood, CA, October 2012.

[12] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, NY, October 2003.

[13] Michael A Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558–570, 2014.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[15] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. TaintPipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th Usenix Security Symposium*, Washington, D.C., August 2015.

[16] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[17] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[18] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '06)*, Orlando, FL, 2006.

[19] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[20] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen,

Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2008.

[21] Benjamin Wester, David Devescery, Peter M. Chen Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, TX, March 2013.