Sonora: A Platform for Continuous Mobile-Cloud Computing

*Fan Yang *Zhengping Qian *Xiuwei Chen †Ivan Beschastnikh *Li Zhuang
*Lidong Zhou *Jacky Shen

*Microsoft Research Asia †University of Washington

Abstract

This paper presents Sonora, a platform for mobile-cloud computing. Sonora is designed to support the development and execution of *continuous* mobile-cloud services. To this end, Sonora provides developers with stream-based programming interfaces that coherently integrate a broad range of existing techniques from mobile, database, and distributed systems. These range from support for disconnected operation to relational and event-driven models. Sonora's execution engine is a fault-tolerant distributed runtime that supports user-facing continuous sensing and processing services in the cloud. Key features of this engine are its dynamic load balancing mechanisms, and a novel failure recovery protocol that performs checkpoint-based partial rollback recovery with selective re-execution. To illustrate the relevance and power of the stream abstraction in describing complex mobile-cloud services we evaluate Sonora's design in the context of two services. We also validate Sonora's design, demonstrating that Sonora is efficient, scalable, and provides responsive fault tolerance.

1. Introduction

The rapid penetration of smart phones is enabling an increasingly popular class of services that *continuously* process context-sensitive information sensed or collected from mobile phones [18, 41]. These services may also correlate and aggregate such information across devices and with other web content on the server, as well as deliver context-sensitive information back to mobile devices. Such services tend to leverage cloud platforms to scalably and reliably store and process large amounts of user generated content. As a result, they promote a close integration between mobile and cloud platforms. This integration creates new technical challenges for effectively supporting continuous mobile-cloud data processing.

Cloud challenges. Continuous mobile-cloud data processing poses unique cloud challenges that are not sufficiently addressed by existing systems. For example, because data arrives continuously and unpredictably, run-time monitoring and adaptation to the dynamics of incoming data become crucial. Such adaptation is not a concern for existing cloud processing engines, which are optimized for batch data processing [16, 22]. Furthermore, because data from continuously operating services accumulates over time, the failure recovery strategy of re-computation no longer applies – restarting an entire computation is unacceptable. The alternative offered by distributed real-time stream processing systems [3, 8, 10] is also not always desirable. These systems are intended for services with stringent real-time requirements. However, many continuous mobile-cloud data processing services are willing to trade-off latency in exceptional cases. For example, better scalability may be achieved with less replication, at the expense of higher latency during failure recovery.

Mobile challenges. Mobile devices impose challenges of their own. Continuous mobile data operations must be energy efficient. Individual techniques, like those that provide power-saving adaptation and optimize mobile-cloud communication are well-known [26, 36]. However, applying these techniques to a mobile-cloud service in a coherent way requires extensive knowledge across disjoint areas like mobile systems

and cloud computing. This creates a significant challenge for mobile-cloud service developers.

To address the above mobile-cloud challenges we present the design, implementation, and evaluation of *Sonora*, a platform for *M*obile-cloud computing *i*n *ST*reams. Sonora provides developers with stream-based interfaces and an efficient, scalable, and fault-tolerant mobile-cloud data processing engine.

Sonora promotes the *stream* abstraction for the following reasons. First, streams successfully unify many data operations in mobile and cloud environments. As detailed in Section 2, continuous data operations can be conveniently expressed using Sonora's stream-based interfaces, and recurring updates appended to a stream can serve as a series of events triggering further computation. Second, with stream abstraction Sonora relieves programmers from writing networking code, which greatly reduces development effort. Finally, streams allow Sonora to easily incorporate a wide range of well-known optimizations. For example, streams integrate techniques to conserve mobile energy, such as data transmission batching, compression, and smart scheduling. In the cloud, the abstraction supports scalable stream operators and provides load balancing with stream-based partitioning and scheduling. Also, techniques in database like multi-query optimization [38, 49] can help improve computing efficiency both on-mobile and in-cloud.

Sonora's cloud execution engine maps a logical data-flow plan into a physical plan, which involves physical resources. The engine is designed particularly to cope with continuously appended data streams and continuous processing. Sonora's cloud processing scales to large amounts of incoming data, and remains responsive to produce results in a timely manner as more data arrive. Sonora monitors execution and shifts load in response to spatial and temporal load imbalances that occur in continuous compute workloads. Sonora provides the high availability necessary to survive machine failures with flexible latency overhead using a novel checkpoint-based recovery mechanism that combines partial rollback recovery and selective re-computation.

We implemented Sonora and present evaluation results for two services built using Sonora. Our first service is a participatory sensing service called PEIR (Personal Environmental Impact Report) [34], and our second service provides access to popular keywords mined from geo-tagged tweets [41] within a radius of a location. These two services indicate that Sonora can accommodate services with different responsiveness requirements. The services also demonstrate the power of the stream abstraction in describing complex mobile-cloud services that may involve large number of mobile devices that impose a correspondingly massive data processing cloud workload. Overall, our evaluation illustrates the efficacy of the design decisions that shaped Sonora.

Contributions. To summarize, our work makes the following contributions. First, we make a case for the utility of the stream abstraction in developing continuous mobile-cloud services. We do so by implementing and evaluating two practical service. Sonora extends the traditional usage of stream in database systems to further support various optimizations in mobile-cloud communications. To the best of our knowledge Sonora is the first to consider mobile-cloud computing as continuous data processing and apply stream abstraction in this setting. Second, we present the design and evaluation of a scalable and fault-tolerant distributed system to support user-facing continuous sensing and processing services in the cloud. Third, we describe and evaluate a new failure recovery protocol that uses a checkpoint-based *partial* rollback recovery mechanism with *selective* re-execution. A key difference in the proposed scheme is, to support better scalability in the *normal* case Sonora requires less replication than traditional fault tolerance mechanisms in distributed stream processing engines such as Borealis and FLuX [5, 39], at the expense of higher latency in *exceptional* cases, i.e., during failure recovery.

The rest of the paper is organized as follows. Section 2 overviews Sonora's stream interfaces. The Sonora system architecture is described in Section 3, followed by an in-depth description of the Sonora cloud runtime in Section 4. We present an evaluation of Sonora in Section 5, survey related work in Section 6, and conclude with Section 7.

2. Programming with Streams

Sonora embraces streams as the unifying programming abstraction for continuous mobile-cloud services. This abstraction captures two perspectives. The first view is *relational* – Sonora organizes the continuously generated data in the form of a stream and provides SQL-like interfaces to operate over the stream. The second view

is *event-driven* [42] – computing is driven with recurring stream updates and data operations that subscribe to stream updates. Sonora incorporates traditional interfaces from stream databases, including stream scoping (windowing) and stream composition [3, 8]. Stream partitioning is also supported for parallel in-cloud computation.

To relieve programmers from writing network codes Sonora introduces the notion of *sync stream*. In addition to the support of traditional operations in stream database systems, sync streams gracefully support disconnected operation [23] and provide common optimizations for mobile-cloud communications, such as batching, compression, and filtering. Moreover, using sync streams programmers can easily move computation modules between the mobile and cloud platforms.

In summary, one contribution of Sonora is that it adopts the stream abstraction to coherently *integrate* a broad range of techniques from mobile, database, and distributed systems areas. Details of Sonora's stream interface are described in the following subsections.

2.1 Basic Stream Operations

Stream construction, subscription, and triggers. A stream is a continuously expanding data sequence with elements ordered according to their timestamps. Line 1 in Figure 1 defines a stream of data type `Location`. A callback function `TripSegmentation` is subscribed to this stream (line 2) and is triggered when a new stream element is appended. A stream can be explicitly appended to (line 3), in this case by reading from the GPS sensor.

A callback can subscribe to more than one stream. With `Zip(X, Y, c)`, the callback `c` is triggered when both `X` and `Y` have a new element (matched as a pair); while with `Combine` the callback executes when either stream has a new element.

Active streams. A mobile sensing application is usually driven by sensor inputs, expressed in Sonora as *active* streams. An active stream is associated with a data source and has an update interval, as shown in lines 5-7 in Figure 1.

Relational stream operators. Sonora has built-in relational support, including filtering, projection, union, grouping, join, and aggregations. A relational operator treats a stream as a relational table containing all elements appended to the stream. Lines 11-13 of Figure 1 define stream `Y` by filtering elements in `X` and then applying the function `MatchLocationOnMap` to each remaining element.

```
1  var trace = new Stream<Location>;
2  trace.Subscribe(TripSegmentation);
3  trace.Append(GPS.Query());
4
5  var gps = new ActiveStream<Location>
6    (GPS.Query, TimeSpan.FromSeconds(30));
7  //...define processing logic based on the GPS stream
8  gps.Activate();
9
10 Stream<Location> X;
11 var Y = X.Where(l =>
12   l.Precision < MAX_ERROR_ALLOWED)
13   .Select(MatchLocationOnMap);
14
15 Stream<double> X;
16 var Y = X.HoppingWindow(TimeSpan.FromSeconds(5))
       .Average();
17
18 Stream<double> X; // temperatures
19 var peakTemperature = State<double>
20  .Aggregate(X, (oldPeak, x) => (x > oldPeak ? x :
       oldPeak));
21
22 Stream<Waypoint> X;
23 X.Partitionable<int>(pt =>
       pt.USER_ID).Subscribe(Func);
24
25 SyncStream<double> X;
26 SyncSinkStream<double> Y = X.GetSink();
27 Y.Subscribe(...);
28 X.Append(...);
29
30 SyncStream<double> X;
31 X.Batch(MAX_ELEMENT_COUNT, TIMEOUT).Zip();
```

Figure 1. Basic stream operations examples.

Windowing. Sonora allows explicit control of when subscribers are triggered and which subset of elements an operator may observe. Time or count based *windowing* is a common way to express this. Lines 15-16 in Figure 1 define stream `Y` as a stream of averages for every 5 second window of values in stream `X`.

Stateful streams. Sometimes only the latest value in a time series matters. *Stateful streams* are used to define a changing state. Lines 18-20 in Figure 1 define a stateful stream to track the maximum (temperature) value observed in stream `X`.

Stream in-cloud partitioning. Developers may annotate streams to direct execution and optimization. For example, stream `X` on lines 22-23 of Figure 1 is annotated as `Partitionable` with a key selector `USER_ID`.

```
1  // Built−in timer stream: ActiveStream<bool>
2  var signals = new
       TimerStream(TimeSpan.FromSeconds(30));
3  // Lower sampling rate when in low battery mode
4  int n = 0;
5  signals = signals
6    .Select(_ => {
7      n++;
8      return (LOW_BATTERY ? (n % N) == 0 : true);
9    })
10   .Where(b => b);
11 // Filter out samples taken indoors
12 signals = signals.Where(!Context.IsIndoor);
13 var gps = signals.Select(_ => GPS.Query());
```

Figure 2. Adaptive sampling rate control.

This allows parallel in-cloud execution of the callback `Func`.

Stream composition. Streams can be composed, either with a series of subscribers or relational operators. The original code in Figure 2 consisted of lines 2 and 13, which create a `gps` stream of GPS readings, and a `signals` stream to control the sampling frequency of the `gps` stream. To lower the sampling rate by a factor of `N` when the battery is low, we added lines 4-10. Line 12 further removes GPS readings when the mobile is detected to be indoors.

2.2 Sync Streams

In addition to the relational and windowing operators that are available in traditional stream databases, Sonora uses *sync streams* for communication between the mobile and the cloud to relieve programmers from writing networking code. A sync stream is maintained in the cloud and on mobile, and is a reliable unidirectional channel between a *source stream* and the *sink stream*. A synchronization policy directs how and when items are propagated from the source to the sink. For two-way communication programmers can use two sync streams.

Lines 25-28 in Figure 1 illustrate basic usage of a source stream `X` and a corresponding sink stream `Y`. Updates to `X` are propagated to `Y` and trigger subscribed callbacks on the sink stream. Synchronization policies offer great flexibility, which we now illustrate with several built-in mobile communication optimizations.

Disconnected operation. Mobile devices often suffer from disconnections [23]. Sonora source streams buffer data during disconnections and resume normal operation upon reconnection. Connectivity interruptions are handled transparently if this is desirable; an application may also decide to be notified when disconnections occur.

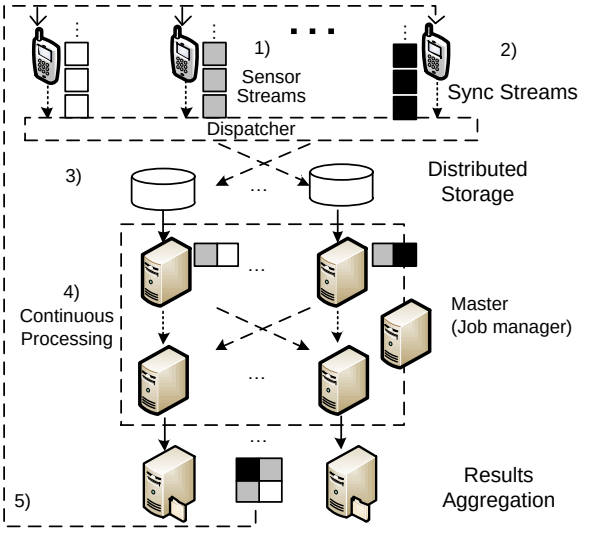

Figure 3. A data-flow view of Sonora.

Batching. For energy efficiency, mobile applications often buffer data (and set the radio into a sleep mode) until enough data has accumulated or until connectivity improves. Sonora supports batching natively and uses it to allow users to trade off power for data freshness.

Compression. Sync streams provide transparent data compression with algorithms like gzip and delta encoding. Programmers may also implement custom compression algorithms. Lines 30-31 in Figure 1 define a sync stream that uses batching and compression.

Filtering. To conserve energy, Sonora evaluates each new element against a programmer-defined filter operator and appends only those elements that pass the check. For example, PEIR uses filters to discard insignificant changes in location by comparing consecutive GPS readings.

With sync stream, programmers can also easily place the same module on the cloud or on the mobile. For more details see Section 5.4.

3. Sonora System Architecture

3.1 A Data-Flow View of Sonora

Figure 3 overviews Sonora from a data-flow point of view. On the mobile, subscribers of active sensor streams process new sensor readings to extract high-level context and semantics (step 1). For example, mobile user's activity (e.g., walking or driving) can be inferred from sensor streams and be used to control

sampling rates of sensor streams for power efficiency: a mobile can reduce GPS sampling rate when stationary or turn off GPS when indoors. Sync streams are then used to stream data from mobiles to the cloud for aggregation and further processing (step 2). In the cloud, streaming data from a large number of mobiles are stored reliably (step 3) and processed in a scalable, fault-tolerant, and efficient manner (step 4). The results are then delivered back to individual mobile devices via sync streams (step 5). These result streams can be considered as special "sensor" streams on the mobile devices and can trigger further processing or their values can be displayed to the users.

3.2 Sonora System Architecture

Figure 4 presents the Sonora system architecture. Sonora includes a programming development kit and execution environments for the mobile and cloud platforms. The programming development kit contains (1) a library with the stream interface and stream support utilities; and (2) a compiler which compiles the original user code into two parts, one for execution on the mobile device and one for execution in the cloud.

The compiler tool takes a user program that uses the Sonora stream interface and performs the following steps.

Data flow graph. First, the program is compiled into a *data flow graph*. Each vertex in this graph is a stream operator. A stream operator receives data from *input* streams and outputs the results of its computation into *output* streams.

Logical plan. The data flow graph is compiled into a logical plan, which combines some consecutive operators together into *executable vertices*. Executable vertices are connected with streams, which represent data flow in the system during execution. Each execution vertex is a stand-alone executable unit that can be assigned for execution on a machine. The logical plan also defines the execution boundary between the mobile and cloud platforms. Developers define this boundary explicitly.

Window optimizations. When generating a logical plan, the compiler applies two window-based operation optimizations. First, instead of performing an operation on individual windows that contain a subset of data elements in the stream, Sonora can sometimes calculate the result of the next window based on the prior one. This is possible when the windows overlap and

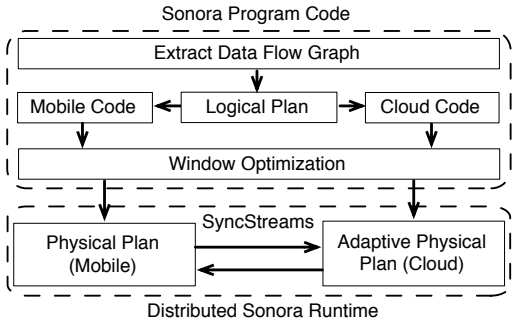

Figure 4. Sonora system architecture.

incremental computation of an operation is possible. We have implemented incremental window operations for common aggregations such as mean, variance, and Discrete Fast Fourier Transformation (DFFT). Second, when *multiple* operators exist, redundant computation may occur. As a further optimization we eliminate such redundant common sub-computation [49].

Physical plan in the cloud. The logical plan subgraph assigned to the cloud is shipped to the cloud runtime. This runtime schedules and assigns computation resources, resulting in a physical plan. This plan contains information like the number of executable vertex instances and machine assignments for each vertex. Each logical vertex is therefore mapped to a set of *physical vertices*. As the incoming stream rate and machine availability change the physical plan is dynamically adapted.

Sonora runtime. Program execution on the mobile device is straightforward. The device runs the executable as a client program. Execution in the cloud is more involved. The executable shipped to the cloud includes a serialized subgraph of the logical plan and a set of binary libraries (e.g. DLLs) of user code. The cloud runtime starts a *job manager* (Figure 3) with the subgraph and binary libraries as input. The job manager is responsible for translating the logical plan into a physical plan, and for assigning physical vertices to machines. The job manager also starts a *dispatcher* service where all stream data from mobile devices is first received and then distributed to a set of corresponding workers.

A job manager further monitors the execution to adjust the physical vertex assignment in response to changes in the system, including load changes and failure. The core of the cloud runtime is a distributed stream query execution engine that is scalable, adaptive, and fault tolerant. We elaborate on its design in the next section.

4. Sonora in the Cloud

Popular services that continuously gather mobile sensor readings amass billions of data points from millions of devices. The cloud is well suited to operating at such scales. Similar to existing batch-oriented data storage and processing systems [16, 19], Sonora stores data reliably and scales to include more machines as the number of users and the rate of incoming data increase. Unlike batch processing systems, Sonora is designed to cope with the continuous nature of data arrival and processing. Sonora *streams* are constantly appended to, and Sonora computation is continuous, not run-once. Because computation is long-lived and incoming data arrives at variable rates, runtime monitoring, dynamic adaptation, and load re-balancing are crucial in *normal* execution. In contrast, a MapReduce-like batch processing engine mostly cares about *outliers* during its execution [2, 16]. Moreover, the MapReduce style of handling failures via re-execution is problematic for continuous computation – re-running computation from the beginning is not viable.

The stream abstraction enables Sonora to leverage work on stream processing engines [3, 8, 10]. However, unlike real-time stream processing, Sonora covers services that do not necessarily have stringent latency requirements. This allows Sonora to trade off latency to gain fault tolerance with less overhead, and better throughput and scalability. Sonora also supports continuous incremental computation over long time-windows (e.g., every day). The absence of real-time constraints allows Sonora to materialize large streams in distributed storage to optimize incremental computation.

In the rest of this section we discuss how Sonora provides different tradeoffs between responsiveness, reliability, and scalability, and how this differentiates it from existing systems.

4.1 Storing and Maintaining Streams

A stream's implementation depends on its use. Sonora can maintain a single stream entry, the last N entries, or all entries in the last M seconds according to the specific interface used to construct the stream. For example, entries in a sliding window might be laid out as a ring buffer. When an entry is accessed by a subscriber based on a key other than the timestamp, it is more beneficial to store the entries in a data structure such as a hash table. This in-memory processing design is optimized for low latency and is shared across mobile and cloud platforms.

The Sonora cloud runtime provides reliable and scalable storage service to store large amounts of stream data for both incoming data streams or output streams as results of a computation, as well as checkpoints. Sonora's storage service is based on PacificA, a scalable storage system [27]. PacificA can partition stream entries across a set of machines and dynamically scale up to involve more machines by re-partitioning the stream as it grows. Each machine storing a partition maintains (1) a "base" portion of the partition on local persistent storage, (2) a series of delta portions on local persistent storage representing updates to the base portion, and (3) an in-memory portion containing the most recent updates to the partition. A centralized partition manager maintains the partition map. For data reliability, each partition is replicated using a replication protocol described in [27], with the partition manager maintaining the replica set for each partition. The partition manager itself is replicated using Paxos [25].

4.2 Fault Tolerance and Failure Recovery

For any physical vertex V in the physical plan DAG, we term all vertices that generate data as input to V as *input vertices* of V, and we term all vertices that receive output from V as *output vertices* of V. V's *downstream vertices* are all vertices whose input is derived from V's output, which includes V's output vertices, their output vertices, and so on, as a transitive closure.

Computation in Sonora is long-running and must survive faults. Failure recovery via re-computation from prior stages is not appropriate for two reasons: (1) restarting a long-running execution from the beginning imposes an unacceptable latency, and (2) streaming data introduces computation non-determinism, complicating replay. Consider a simple reduce-like phase, where a computation vertex accepts incoming data from multiple vertices in the previous stage. The interleaving of data from source vertices introduces non-determinism. A typical map/reduce computation avoids this problem because a reduce is performed on all data across all source vertices. It is possible that certain computation is insensitive to the order of incoming data. However, even in this case, the state of a vertex may depend on how much source vertex data was received and processed.

Sonora checkpoints computation periodically. It adopts a combination of *partial* rollback recovery and *selective* re-computation from previous stages to strike a balance between run-time overhead and recovery cost. This is different from existing map-reduce style re-execution as well as fault tolerance mechanisms in distributed stream database systems.

The incoming data from mobile devices are first logged in PacificA, Sonora's reliable data store. These data logs are eventually garbage collected when their corresponding end results have been reliably recorded.

Consistent global checkpointing. Sonora uses developer-assisted checkpointing to extract and store application state. The checkpointing process is similar to Chandy-Lamport's snapshot protocol [11], but is customized for DAG-like data flow computation in Sonora.

All checkpoint data are written to PacificA, which performs replication for data reliability. Note that checkpointing is *not* on the critical path of computation: normal computation does not have to wait for checkpointing to complete before proceeding.

We do not claim any contribution to the checkpointing algorithm. The following description sketches the protocol in a level that is sufficient for readers to understand the mechanism in the context of Sonora.

Checkpointing starts when a central controller dispatches checkpoint markers to the incoming source vertices. Each vertex V uses an array `received` for tracking input vertices from which the markers have not arrived. V must record all messages as part of the checkpoint from those input vertices until the markers arrive. The state of a checkpoint on each vertex transitions from `INPROGRESS`, to `COMPLETED`, and then to `STABLE`. When the state is `STABLE` on V, that same checkpoint must be `STABLE` on all its input vertices. When a checkpoint is `STABLE` on all vertices in the last stage, it is `STABLE` on all vertices. We use this condition for garbage collection purposes.

More exactly, checkpointing on vertices is performed as follows:

(i) A central controller inserts checkpoint markers with the next highest checkpoint version number c into all input data sources to initiate a new checkpoint c.

(ii) Upon receiving a marker c from input vertex I, a processing vertex V checks whether this is the first marker c it has received. If so, it executes step (ii.a); otherwise, it proceeds to step (ii.b).

(ii.a) V records its local state as part of the checkpoint, and inserts the marker c on its outgoing channels to all output vertices. It initializes array $\texttt{received}[c][N]$ to `false` for every input vertex N. It further sets array $\texttt{State}[c]$ to `INPROGRESS`.

(ii.b) V sets $\texttt{received}[c][I]$ to `true`. If $\texttt{received}[c][N]$ is `true` for every input vertex N of V, it sets $\texttt{State}[v]$ to `COMPLETED` and notifies all its output vertices of the completion of checkpoint c on vertex V.

(iii) Upon receiving a data message m from input vertex I, if $\texttt{State}[c] = \texttt{INPROGRESS}$ and $\texttt{received}[c][I] = \texttt{false}$ holds, the vertex records (I, m) in the checkpoint.

(iv) Upon receiving notifications from all its input vertices of the completion of checkpoint c on those vertices, the vertex sets $\texttt{State}[c]$ to `STABLE`.

Output recording for selective re-execution. Like map/reduce, a vertex records its output *locally*. This allows for re-execution that selectively starts from the latest checkpoint when any of its output vertices fail. We do not record such outputs in PacificA (as with checkpoints) because of the latency penalty. If such output is lost, the system can fall back to an earlier stage for re-execution and in the worst case re-read the reliably stored mobile input to the computation.

This is another different design decision compared to those fault tolerance mechanisms of parallel stream databases like FLuX, SGuard, and Borealis [5, 24, 39], which use a higher degree of replication to reduce latency during failure. In these systems, checkpoints, input/output of each operator, and even the run-time operators themselves are replicated across multiple machines. Due to the relaxed latency constraints, Sonora only replicates the checkpoints and the initial mobile input data. As a result, for the same number of machines Sonora has a higher system throughput and similar latency during executions with no failures. However, Sonora's failure recovery latency is longer than that of parallel stream database systems.

Partial rollback recovery. Given that checkpoints are globally consistent, a simple way to recover from failures is to roll back to the latest checkpoint and continue from there. To avoid unnecessary rollbacks, Sonora's recovery mechanism instead uses a partial rollback mechanism. The key observation is that in a DAG computation, a fault at vertex V affects only V's downstream vertices. Therefore, only V and V's downstream vertices need to roll back. After rolling back to the lat-

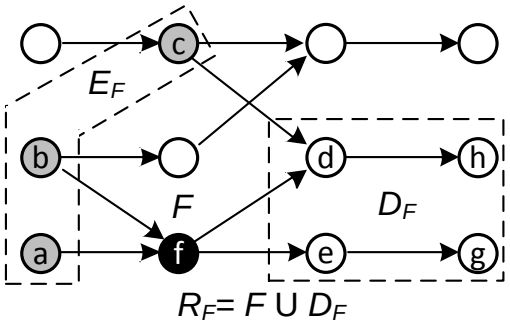

Figure 5. An example of failure recovery.

est stable checkpoint, the recovery of V can be done with re-execution, as in map/reduce, since inputs to V's computation are available in the local storage of V's input vertices.

More precisely, let F be the set of vertices that have failed and D_F be the set of downstream vertices from any vertex in F. Then the rollback set R_F is the union of F and D_F: these vertices will be rolled back to the latest consistent checkpoint. The re-execution set E_F contains all input vertices of R_F that are not in R_F. To restart vertices in R_F from the checkpoint, vertices in E_F must replay inputs. If the recorded output for re-execution is lost on a vertex v in E_F, vertex v has to be added to the rollback set and the re-execution has to start from an earlier stage. In the worst case, all vertices are in the rollback set and the entire computation restarts from that latest consistent checkpoint. Correctness of recovery follows directly from global consistency provided by the global checkpointing protocol, as a special case of the Chandy-Lamport's snapshot protocol.

Figure 5 shows an example of failure recovery. Each circle represents a vertex, with directed edges indicating the direction of data flow. Vertex f is the only failed vertex: $F = \{f\}$. Vertices d, e, g, and h are the downstream vertices of f: $D_F = \{d, e, g, h\}$. All vertices in $R_F = \{d, e, f, g, h\}$ will be rolled back to the latest consistent checkpoint. The shaded vertices in $E_F = \{a, b, c\}$ will replay the outputs to those who are rolling back.

It is worth noting that all downstream vertices in a DAG must be rolled back, along with vertices that actually fail. This is due to non-determinism: when rolled back, re-execution on a vertex could deviate from the previous run due to non-determinism in the computation (e.g., the order in which it processes data from different input channels.) We therefore rollback all downstream vertices that depend on outputs from the re-executions. If the computation is known to be deterministic in certain stages, we can avoid rolling back certain vertices.

To address non-determinism, parallel stream database systems usually serialize and replicate incoming data from multiple input vertices for future deterministic replay [5]. This improves the recovery speed but imposes a serialization and replication burden on the system in the critical execution path of the normal case.

Garbage collection. With globally consistent checkpoints, garbage collection becomes straightforward. When a checkpoint of version c becomes stable at the end of the computation, data necessary to reproduce c can be removed. This includes all logged incoming data up to the checkpoint marker for c, all lower-version checkpoints, and the output recorded for execution at intermediate stages up to the checkpoint. In practice, to guard against catastrophic failures it might be prudent to maintain multiple checkpoints in case the latest one is lost or damaged. This is especially the case when there is ample cloud storage.

Job manager fault tolerance. A job manager is responsible for monitoring the execution of a job and maintains the mapping from a logical plan to a physical deployment. Because a job in Sonora is long-running, a job manager must be made fault tolerant. A simple solution is to have a job manager write its state to reliable storage. Because its state is updated infrequently, this is not on the critical path of job execution and the performance impact is minor. A central master (which itself could be replicated using state machine replication) can monitor the liveness of job managers and assign a new machine to take over when a job manager fails. The new job manager would then load the state from reliable storage and assume the new role. Leases [20] from the central master can be used to ensure that there is only one active job manager for each job, as done in Boxwood [29] and BigTable [12]. Sonora has yet to include this feature in the current implementation.

4.3 Dynamic Load Adaptation

In a continuous mobile-cloud service (e.g., PEIR), data processing is generally carried out in multiple stages. Potentially this could be represented as one or multiple map/reduce jobs if all data were available at the start of the computation. However, Sonora supports applications where data arrives continuously and computation

is triggered as new data become available. Such continuous computation may create two types of imbalances. The first type is *spatial imbalance* – a certain portion of the system may become overloaded. Spatial balance is non-trivial to maintain for continuous computation as characteristics of incoming data could change over time. The second type of imbalance is *temporal imbalance*, which may manifest as load fluctuation because of sudden surges in load. Sonora employs load re-balancing and flow control to mitigate spatial and temporal imbalances.

Load re-balancing. In Sonora the *job manager* is in charge of dynamically mapping the logical plan into a physical plan. The job manager monitors the load on spare machines and those involved in the computation. When a machine in a certain stage is overloaded, the job manager recruits a spare machine to take over half of the load. Similarly, if the load on a certain machine diminishes, load re-balancing merges load across some machines. Sonora implements a simple dynamic load re-balancing strategy that suffices for many continuous mobile-cloud services like PEIR.

Continuous mobile-cloud services aggregate collected sensor data across users (e.g., location), we use hash partitioning on user IDs and a dynamic hash function for adaptation. The job manager monitors CPU utilization on each machine and makes load balancing decisions. A machine is considered overloaded when its CPU utilization exceeds a threshold (90% in implementation). When this occurs, the job manager splits the hash range assigned to the overloaded machine and re-assigns half of the range to a spare machine. Hash ranges can also be merged when machines are underutilized (e.g. under 30%). Because the amount of state kept in each processing stage is small for services like PEIR, the overhead of such hash merging and splitting is small. Merging and splitting can also be done with other partitioning mechanisms with varying bookkeeping overhead. This simple scheme is similar to the load balance design in stream database systems like Eddies and FLuX [31, 39].

Flow control. When the incoming data rate suddenly exceeds the processing capacity Sonora uses flow control to cope with the load. The idea is simple – temporarily trade off latency for throughput and scalability. Rather than sending data along a processing pipeline, Sonora's flow control detects overload and redirects data into PacificA. This data is processed once the system catches up with the load. Redirection helps Sonora take advantage of periods of low load for catching up, and enables batching for higher throughput. In case the incoming data rate exceeds the throughput of the storage system, the cloud pushes back onto the mobile devices to adaptively reduce the sensor sampling frequency (see Figure 2 and Section 2.1 for details).

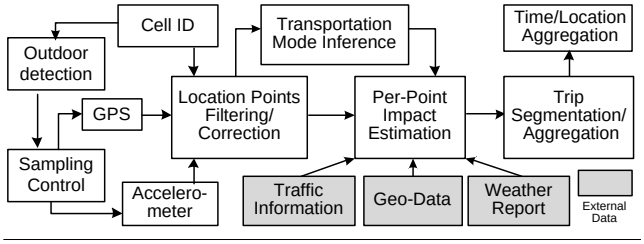

Figure 6. PEIR overview.

Although earlier stream processing engines like Telegraph [10] and Aurora [8] support redirecting streams to persistent storage, more recent real-time stream systems do not adopt such a flow control mechanism due to real-time constraints – the extra time incurred by data redirection may render the results useless to latency sensitive services. Instead, such systems usually rely on load shedding to deal with overload, i.e., selectively dropping some of the incoming data [31].

5. Evaluation

The focus of our evaluation is on the mechanisms presented in the prior sections as well as on their overheads. Unless stated otherwise, we present results for a service implemented on top of Sonora that is modeled after the Personal Environment Impact Report (PEIR) [34]. Figure 6 illustrates the data-flow within our PEIR service. PEIR is a participatory sensing [7] service that continuously aggregates information from mobile devices, computes on this data, and delivers the resulting context-sensitive information back to mobile devices. PEIR is a good example of the kinds services Sonora is designed to support. Next we describe PEIR in more detail.

The PEIR [34] service estimates personalized environmental impact and exposure using sensor data collected from mobile devices. This report includes various *impact calculations*, such as carbon footprint, particle matter emission, and the likelihoods of *exposure* to particle matter and unhealthy fast food. Figure 6 illustrates the parts of PEIR that we designed and implemented with Sonora. Devices participating in PEIR continuously collect sensor data, such as GPS and ac-

celerometer readings, and IDs of cellular towers associated with mobile devices. These data are used to infer user context, such as whether or not the user is outside, and their transportation mode (using a Hidden Markov Model). This context may in turn be used to control sampling. Inferred transportation mode, when combined with weather, traffic, and geo-data, is used to estimate various impacts, following the Emission Factors model (EMFAC) [1]. The location way-points from mobiles are grouped into trips, the primary reporting unit for the impact report. The results are further aggregated based on users and locations (e.g., across sensitive sites such as hospitals and schools).

5.1 Experimental Setup

Our experiments use GPS traces from the GeoLife project [47, 48]. This dataset contains a total of more than 20 million location points, which make up location traces collected from 165 users over two years covering a wide range of outdoor movement. We use randomly generated data to simulate traffic, weather conditions, and related geo-data.

We report results for experiments performed on an HTC Touch Diamond 2 running Windows Mobile 6.5 with .NET Compact Framework 3.5. During all experiments the phone used its native WiFi interface for communication. In the experiments our cloud platform is a cluster of 40 machines, each with a dual Intel Xeon CPU X5550 (2.67GHz), 12GB RAM, two 1.0TB SATA disks, running the 64bit version of Windows Server 2008 and .NET framework 3.5. These machines use a 1Gb Ethernet interconnect. For collecting measurements we used the built-in Sonora logging tool. We used the system call `GetSystemPowerStatusEx2` to query the power status of the mobile phone every second for power measurements. The obtained power measurements are typically higher than the actual value, but they can be used for a meaningful comparison.

5.2 Sonora Cloud Runtime Performance

Scalability. Figure 7 plots the CPU usage and network throughput for a server machine processing PEIR location data from mobile devices. CPU usage increases faster than throughput and the CPU is overloaded before the 1Gbps network is saturated. This indicates that our implementation of PEIR is CPU-bounded. We therefore measure and report Sonora scalability in terms of location points Sonora cloud runtime can process per second.

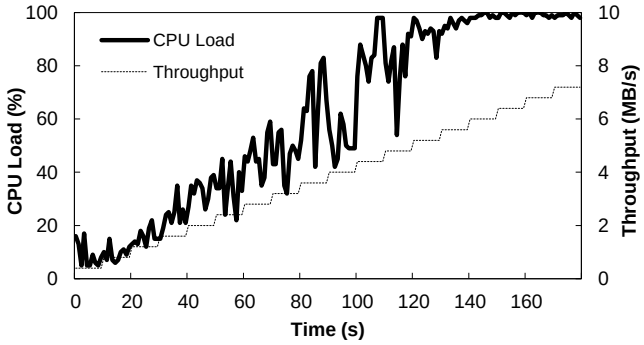

Figure 7. PEIR is CPU-bounded.

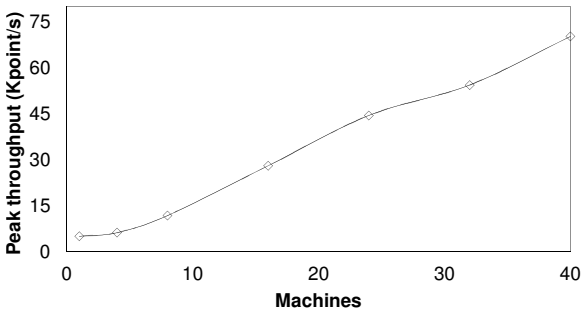

Figure 8. Sonora scalability.

Figure 8 shows that the Sonora implementation of PEIR scales linearly – peak throughput increases proportionally to the number of machines. The speedup efficiency is 0.70 when the number of machines is 40. If each mobile user reported their location once every 5 seconds, Sonora would be able to support over 350,000 users *concurrently* with just 40 machines. In addition, sync stream filters can reduce the amount of traffic from mobile devices by filtering out insignificant changes in location. This would allow Sonora to support even more users.

Load balancing. We varied the incoming data rate to evaluate the efficacy of Sonora's online load balancing mechanism. Figure 9 shows that the Sonora cloud runtime (1) is able to keep track of resource usage in the cloud platform, and (2) assigns a proper number of machines in response to a change in the data rate in a timely manner.

Figure 10 shows the CPU load distribution across 19-node and 38-node configurations. As discussed in Section 4, a Sonora policy constrained CPU load to range between 30% and 90%. The Figure shows that only 3 out of the 57 machines (5%) were outside of this range. Log inspection revealed that the 3 machines

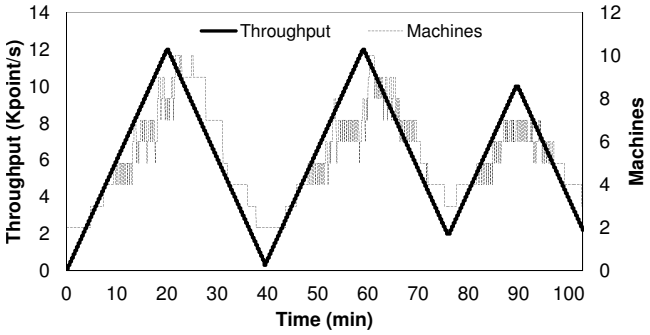

Figure 9. Online load balancing.

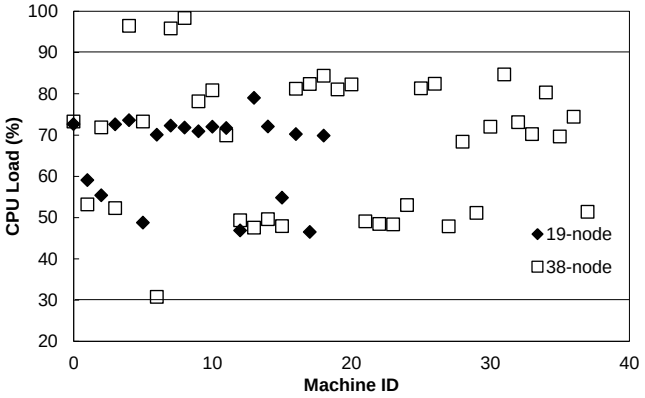

Figure 10. CPU load distribution.

were overloaded because there were no additional idle machine (with CPU load below 30%) to share the load.

Fault tolerance. To evaluate how Sonora recovers from machine failures we turned off dynamic adaptation and ran PEIR on 32 machines, with 9 machines running a physical vertex in the first stage and 23 machines running a physical vertex in the second stage. This assignment is balanced as the second stage is more costly than the first. All machines in the first stage generate data that is consumed by all machines in the second stage. A dispatcher was used to feed input to vertices in the first stage.

In the first experiment, illustrated in Figure 11, we set the checkpointing interval to 40 seconds. We took two checkpoints (at 20s and 60s) and failed a machine at 80s that was responsible for a vertex in the first stage. In experiments we introduce faults halfway through a checkpointing interval – 20 seconds in this case. The recovery protocol mandates that all 23 vertices in the second stage, along with the failed vertex in the first stage roll back. During this recovery the remaining 8 vertices and the dispatcher replay their output.

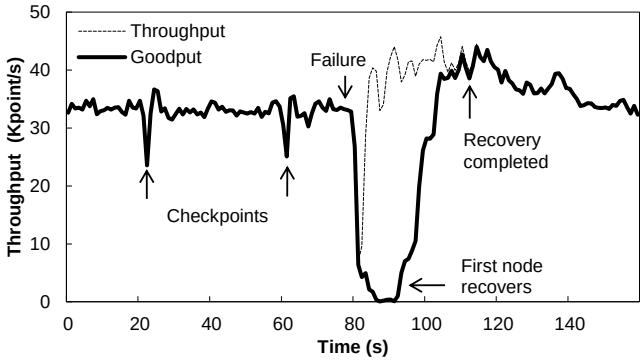

Figure 11. Throughput and goodput over time as Sonora checkpoints twice and then recovers from a failure.

Because rollback recovery will re-produce outputs that are lost due to failures, Figure 11 plots both the system throughput aggregated across all stage-two vertices along with the user-observed *goodput*, which does not include the outputs reproduced during recovery.

For simplicity, our current checkpointing process first flushes all buffers to avoid recording them. This stalls the pipeline, which translates into the noticeable throughput drop during checkpointing. This is not inherent to our design, but is a consequence of the simpler implementation. Optimized checkpointing implementation using copy-on-write technique could amortize the throughput decrease. In the experiment it took about 11 seconds for each checkpoint to become stable across all vertices. The checkpoint size was approximately 8.8 MB (and three copies were stored).

During recovery, because there are other stage-two vertices generating results, the goodput does not immediately drop to zero: we flush these buffers before rolling back. Goodput is zero for 8 seconds until recovery completes on the stage-two vertices and new outputs are generated. Recovery completes across all vertices 36 seconds after the failure. Because the input rate does not saturate the system, goodput is above average right after recovery due to buffered data processing. The recovery generated a total of 266.12 MB of network traffic, of which 53.27 MB were used to replay output logs and the remaining 212.85 MB were used for checkpoint loading.

To evaluate Sonora's checkpointing overhead we used the same setting as above and varied the checkpointing interval from 40 to 120 seconds. In each case we introduced a fault in the middle of the checkpointing interval. Table 1 shows that while the checkpoint

Interval (s)	Size (MB)	R-Cost (MB)	R-Time (s)	RC-Time (s)
40	8.87	266.12	8	36
60	8.78	288.27	12	50
80	8.78	317.27	17	64
100	8.78	321.58	23	76
120	8.87	355.60	31	88

Table 1. Failure recovery cost for different checkpointing intervals. Interval: checkpointing interval; Size: checkpoint size; R-Cost: the network traffic cost of failure recovery; R-Time: earliest recovery time; RC-Time: recovery completion time.

Day	Process time(s)		
	8-node	4-node	2-node
1	15.1	12.0	20.4
2	12.2	30.5	18.7
3	17.7	10.6	17.9
4	21.9	19.8	35.8
5	13.0	20.2	19.2
6	10.8	14.4	23.2
7	17.7	21.0	23.3
Overall time	111.2	128.0	159.1
Unoptimized overall time		296.7 (8-node)	

Table 2. Processing time for a 2GB daily tweet stream.

size is stable, a longer checkpointing interval causes a more expensive recovery. This manifested as a longer recovery time and a higher network cost. Results in Table 1 also support the conclusion that a fault tolerance strategy that restarts the computation from the very beginning, as in Map/Reduce, is undesirable in this setting.

Finally, we compared recovery with partial and full rollback for a checkpointing interval set to 120 seconds. Measurements for full rollback give us an upper bound on recovery cost, which would be the observed cost if Sonora used a standard rollback recovery mechanism. Our results indicate that partial rollback incurred 63% less network traffic and had similarly benefited recovery time. Certainly, the exact benefit of partial rollback depends on the failure scenario and the structure of the computation. In this experiment, we were already rolling back 24 of the 32 vertices. Even so, partial rollback incurred significantly less overhead than full rollback because the raw input to stage one vertices was much larger than the processed input to the vertices in stage two.

Incremental computation on streams with large time-windows. PEIR aggregates location data over small window sizes for individual users. In this experiment we use another service to demonstrate Sonora's ability to handle incremental computation on streams with large time-windows. The service extracts the most popular keyword from all geo-tagged tweets [41] within a geographic distance over the last 7 days. The calculation is performed daily.

In the experiment, we assumed a 33x33 grid preseeded randomly with a random distribution of 2GB of daily tweet data. Each tweet is 5-20 words long, with word selection following the word frequency of an English novel [43]. Although this setting does not reflect real world usage, it is nevertheless useful for evaluating large stream processing enabled by Sonora.

We use an `updates` stream to store nearby tweets over the last 7 days. To avoid large aggregate computation, the keyword popularity calculation is divided into each day and is updated daily. Note that this is different from PEIR in which the computation is triggered whenever new location data is available. In the implementation we use a `dailyUpdate` stream to tally the keywords for each day, similar to materialized views in databases. All these output streams are partitioned and indexed by location. Each element in the `updates` stream therefore matches up with 7 entries in the `dailyUpdate` stream. The storage system partitions the `updates` stream across multiple machines so that the keyword calculation can execute in parallel. This is similar to a map/reduce job except that the operation repeats everyday with the updated tweets from the `updates` stream.

Table 2 shows the processing time of daily results over 7 days with 8, 4, and 2 Sonora machines. Due to materialized views of daily results, the aggregated processing time with materialized views is much smaller than that without them – for 8 machines the difference is more than factor of two (111s versus 297s). The query latency for the mobile phone to receive the calculation result is just 2.8 second in the 8-machine case. As expected, as the number of cluster machines is reduced from 8 to 2, the overall processing time increases.

Accel.	GSM	Battery	GPS	Fixed	Adaptive
0.27	0.28	0.19	0.88	1.00	0.70

Table 3. Power consumption in Watts when reading different on-phone sensors (first four columns), and when reading all sensors with Fixed and with Adaptive sensing strategies (last two columns).

Power Consumption (Watt)			Network Traffic (MB)	
None	Batch	Batch+Zip	Batch/None	Zip
1.35	0.50	0.45	2.087	0.392

Table 4. Impact of different sync stream optimizations.

In summary, with the help of scalable storage system Sonora is able to process large streams that might not fit in memory.

5.3 Stream-Based Optimizations

Energy efficient adaptive sampling. Adaptive sampling is a useful optimization for saving power in special circumstances. It is straightforward to employ the Sonora stream API to express adaptive sampling. To see this, consider a mobile scenario in which the phone is either indoors or outdoors. The first four columns of Table 3 list the average power consumption (over 5 runs) of a stream operator querying an on-phone sensor at the rate of once every 5 seconds over a period of 5-minutes. The table indicates that the GPS is expensive to read. Indoors, however, an application can save power by disabling GPS, which cannot receive signals anyway. To infer whether the device is indoors or not a lower power sensor like GSM can be used to correlate associated GSM tower ID history with the GPS signals.

To gauge the impact of this adaptive sensing strategy on power consumption we conducted a 30-minute experiment in which the mobile phone was indoors for half the time. With the fixed sensing strategy the device read the GPS, accelerometer, and the battery sensors once every 5 seconds (total power consumption is shown in the Fixed column of Table 3). With the adaptive strategy the device used GSM tower associations to determine whether it was outdoors or indoors (Figure 2 lists the code). When the device detected itself to be outdoors, a stream operator read the GPS, accelerometer, and battery sensors once every 5 seconds. When the device detected itself to be indoors, the operator read just the GSM, and battery sensors at the same frequency. The total power consumption is shown in the Adaptive column of Table 3.

Power measurements for the two strategies (Fixed and Adaptive) indicate that adaptive sensing can save 30% of energy in this setting. Note that adaptive sensing has a small overhead even if the device were to be outdoors at all times. This is because GSM is usually an always-on hardware on a mobile phone, and the algorithm to detect indoor status has low CPU usage.

Sync stream. Sonora relieves developers from optimizing mobile-cloud communication by providing a sync stream abstraction. Table 4 shows the impact of various sync stream optimizations for a HTC phone during a 30-minute run where a sync stream was used to transmit GPS data from the phone to the cloud at a rate of 1Kb/s. Without any optimizations the phone consumed the most energy. When the sync stream batched GPS data into a single transmission every 1 minute, the phone saved 60% power. Although compression did not save much power compared to batching, it did decrease network traffic from 2.087MB to 0.392MB – an 81.2% improvement. All of these optimizations are transparent to the programmer, and can be enabled adaptively.

Sync streams also help applications gracefully handle disconnections. Figure 12 plots the total KB transmitted from a phone to the cloud using a single sync stream over time. When a disconnection occurs at time t_0, network throughput drops to zero. The sync stream buffers the location data during the 1 minute disconnection and uploads the buffered data in a burst after the phone reconnects to the cloud at t_1.

Our intent with these measurements was not to identify an optimal point: tradeoffs could change significantly due to changes in hardware (e.g. CPU frequency), and context (e.g. mobile versus stationary). Instead, these experiments illustrate how Sonora can enable different configurations to be expressed with ease, and how Sonora can help programmers understand and identify the best tradeoffs for their scenario.

5.4 Optimizing Mobile-Cloud Computation

Sonora's use of the same stream interface on the mobile and in the cloud makes it convenient to attempt different placements of modules between the two platforms. For instance, in one section of the PEIR code a change to fewer than 10 lines of code sufficed to move a transportation mode inference module from the mo-

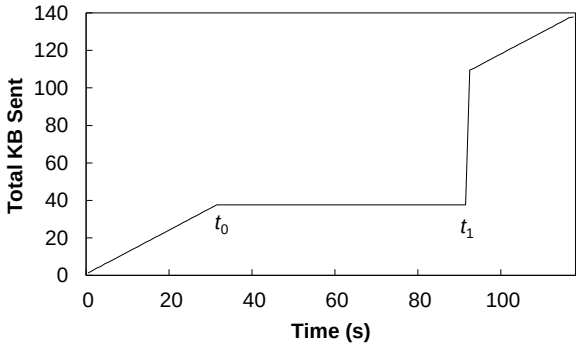

Figure 12. Total KB sent over time with a sync stream. A disconnection is masked between t_0 and t_1.

```
1  // OLD code (all on mobile):
2  Stream<Sample> samples = ...;
3  Stream<double> speeds = samples.Select(...);
4
5  // NEW code:
6  // on mobile:
7  Stream<Sample> samples = ...;
8  SyncStream<Sample> mobOut = new
       SyncStream<Sample>();
9  samples.Subscribe((Sample x) => mobOut.Append(x));
10 // in cloud:
11 SyncSinkStream<Sample> cloudIn =
       mobOut.GetSink();
12 Stream<double> speeds = cloudIn.Select(...);
13
14 // Transportation mode inference ...
```

Figure 13. Code changes to move transportation mode inference from the mobile and into the cloud.

bile into the cloud. Lines 2 and 3 in Figure 13 show the original code that was executed on the mobile. Adding a sync stream, subscribing the sync stream to the sensor readings, and having the cloud receive the readings took four additional lines (lines 8-12) and moved the execution of the following transportation mode inference code into the cloud. In our experience we found that this example generalizes – the stream interface encouraged us to explore different module configurations because of the low programming effort.

Moving transportation mode inference to the cloud reduces mobile CPU utilization, leading to power saving. But this move doubles the network bandwidth because the cloud now also needs to receive accelerometer, GSM signal, and battery status measurements. In aggregate, however, power consumption was reduced by 40% due to the optimizations implemented in the sync stream. Note that this improvement is made at the expense of increased server load.

We also observed that an identical optimization may have different effects on the mobile and in the cloud. For instance, we used incremental computation and common subcomputation elimination to optimize the Discrete Fast Fourier Transform in PEIR's transportation mode inference. On the mobile this reduced power by less than 5% – the low sampling frequency (once every 5 seconds) left the CPU idling for most of the time. Due to aggregation, the gain was much more significant when the optimization was performed in the cloud – peak throughput of a single machine increased by over 200%.

Module placement involves understanding the tradeoffs between mobile energy efficiency, service response time, server load, and other factors. Sonora greatly simplifies the process of identifying these tradeoffs between the two platforms. In the future we hope to outfit Sonora with task partitioning algorithms to repartition tasks as conditions change.

6. Related Work

Sonora was influenced by distributed execution engines in data centers, such as MapReduce [16] and Dryad [22], which process large amounts of data. The Sonora stream interfaces also contains elements of high level declarative languages such as SCOPE [9], DryadLINQ [45], and Pig Latin [37]. These systems are optimized for batch processing and throughput. Instead, Sonora targets continuous mobile-cloud services, often with some latency requirements. The corresponding different design implications are elaborated in details in Section 4.

Recently there are also efforts to "streamline" the computation between different stages in the MapReduce/Drayd framework. These include Hadoop Online [13], HaLoop [6], Naiad [32], and Spark [46]. While the focus of these projects is on enabling pipelining between stages in a batch job, the mechanism is also shown to be useful for continuous queries. Sonora addresses two problems that are also deemed important in that paper, namely, dynamic adaptation and failure recovery. Sonora's failure recovery mechanism is different, partly due to our focus on continuous mobile-cloud services. Continuous bulk processing (CBP) [28] focuses on stateful bulk processing for incremental an-

alytics, where each execution is incremental. It is not intended for interactive applications.

Yahoo! Pipes [44] is a web service that provides programmability over multiple streams with a declarative language interface. Unlike Yahoo! Pipes Sonora targets continuous services. Sonora also uses stream optimizations for efficient mobile-cloud communication.

Twitter's Storm [40] is a cloud-based real-time processing engine with the similar goal as Sonora. However, Sonora provides a different fault tolerant mechanism taking the mobile-cloud computing environment into account. Furthermore, Sonora makes a case that stream is also a useful abstraction in developing mobile programs.

The stream abstraction in Sonora is related to prior work on continuous query processing in databases and sensor networks, including Telegraph, Aurora, TinyDB [3, 8, 10, 30]. There are also commercial products available like IBM System S (a.k.a. InfoSphere Streams) and Microsoft StreamInsight [21, 33]. From these systems Sonora borrows the notion of window, along with common optimizations such as incremental computation and sub-expression elimination [49]. Sonora further applies the notion of streams to mobile platforms and augments the stream abstraction with the sync stream interface to improve mobile-cloud communication, which has been studied extensively in the mobile research community [23, 26].

Many parallel stream database system like Borealis [5], Telegraph's FLuX [39], and SGuard [24] also provide load balancing and failure recovery. These systems are usually engineered to meet a stringent latency requirement. Sonora instead targets continuous processing that does not necessarily have such requirement and allows a different latency/throughput/scalability tradeoffs. This is reflected in Sonora's mechanisms, such as redirecting incoming data to reliable storage to absorb load fluctuation and using re-computation to perform non-deterministic replay for failure recovery, as discussed in detail in Section 4.

Sonora's data-driven architecture is similar to SEDA [42] where applications consist of a network of event-driven stages connected by explicit queues. SEDA also makes use of a dynamic resource controller to keep stages within the appropriate operating regime despite fluctuations in load. However, the SEDA architecture is generally intended for scalable Internet services while the Sonora architecture targets efficient stream processing across the mobile and cloud platforms. Other event-driven systems with publish/subscribe mechanisms include LIME [35] and Limbo [15]. These two systems provide event triggering on top of a tuple space and do not provide a stream abstraction. Sonora's event-trigger programming interfaces also leverage features in functional programming languages.

Systems like Spectra [17], Chroma [4], and the more recent Maui [14] dynamically partition and offload computation tasks from a mobile device to a more powerful server or cloud. Sonora supports execution partitioning, however, this is less of a focus in Sonora and is currently left as a decision to programmers.

7. Conclusion

Mobile and cloud computing are two emerging trends that are likely to reshape computing in the coming years. The convergence of the two is inevitable. Sonora is a distributed platform that embraces this convergence, with a focus on supporting an emerging class of services that perform continuous data-oriented mobile-cloud computing. To ease development of such services Sonora exposes a simple and uniform stream abstraction that coherently incorporates mechanisms from mobile, database, and distributed systems. To cater to the spectrum of latency/scalability requirements that characterize these services Sonora introduces innovations in fault-tolerance, adaptation, and scalability.

References

[1] –. EMission FACtors model, April 2010. `http://www.arb.ca.gov/msei/onroad/latest_version.htm`.

[2] ANANTHANARAYANAN, G., KANDULA, S., STOICA, A. G. I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in map-reduce clusters using mantri. In *OSDI* (2010).

[3] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. Models and issues in data stream systems. In *PODS* (2002).

[4] BALAN, R. K., SATYANARAYANAN, M., PARK, S. Y., AND OKOSHI, T. Tactics-based remote execution for mobile computing. In *Mobisys* (2003).

[5] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S. R., AND STONEBRAKER, M. Fault-tolerance in the Borealis distributed stream processing system. *ACM Trans. Database Syst. 33*, 1 (2008), 1–44.

[6] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. Haloop: Efficient iterative data processing on large clusters. In *VLDB* (2010).

[7] BURKE, J., ESTRIN, D., HANSEN, M., PARKER, A., RAMANATHAN, N., REDDY, S., AND SRIVASTAVA, M. Participatory sensing. In *World Wide Sensor Web Workshop, Sensys* (2006).

[8] CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Monitoring streams: A new class of data management applications. In *Proc. VLDB* (2002).

[9] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[10] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR 2003* (2003).

[11] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (1985), 63–75.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. BigTable: a distributed storage system for structured data. In *OSDI* (2006), pp. 205–218.

[13] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce online. In *NSDI* (2010).

[14] CUERVO, E., BALASUBRAMANIAN, A., KI CHO, D., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making smartphones last longer with code offload. In *Mobisys* (2010).

[15] DAVIES, N., WADE, S. P., FRIDAY, A., AND BLAIR, G. S. Limbo: a tuple space based platform for adaptive mobile applications. In *ICODP/ICDP '97* (1997).

[16] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI'04* (2004).

[17] FLINN, J., PARK, S., AND SATYANARAYANAN, M. Balancing performance, energy, and quality in pervasive computing. In *ICDCS* (2002), IEEE Computer Society.

[18] FOURSQUARE. foursquare, 2010. `http://foursquare.com`. Accessed May 7, 2010.

[19] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. *SIGOPS Oper. Syst. Rev. 37*, 5 (2003), 29–43.

[20] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989), pp. 202–210.

[21] IBM. InfoSphere Streams. `http://www-01.ibm.com/software/data/infosphere/streams/`. Accessed March, 2012.

[22] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41*, 3 (2007), 59–72.

[23] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the coda file system. In *SOSP* (1991), ACM.

[24] KWON, Y., BALAZINSKA, M., AND GREENBERG, A. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow. 1*, 1 (2008), 574–585.

[25] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[26] LI, D., AND ANAND, M. Majab: improving resource management for web-based applications on mobile devices. In *MobiSys* (2009), ACM.

[27] LIN, W., YANG, M., ZHANG, L., AND ZHOU, L. PacificA: Replication in log-based distributed storage systems. Tech. Rep. MSR-TR-2008-25, Microsoft Research, Feb 2008.

[28] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *SoCC* (2010), pp. 51–62.

[29] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI* (2004), pp. 8–8.

[30] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. The design of an acquisitional query processor for sensor networks. In *SIGMOD* (2003), ACM.

[31] MADDEN, S., SHAH, M., HELLERSTEIN, J. M., AND RAMAN, V. Continuously adaptive continuous queries over streams. In *SIGMOD* (2002), ACM.

[32] MCSHERRY, F., ISAACS, R., ISARD, M., AND MURRAY, D. Naiad: The animating spirit of rivers and streams. In *SOSP* (2011).

[33] MICROSOFT. Microsoft StreamInsight. `http://msdn.microsoft.com/en-us/library/ee362541.aspx`. Accessed March, 2012.

[34] MUN, M., REDDY, S., SHILTON, K., YAU, N., BURKE, J., ESTRIN, D., HANSEN, M., HOWARD, E., WEST, R., AND BODA, P. PEIR, the personal environmental impact report, as a platform for participatory sensing systems research. In *Mobisys* (2009).

[35] MURPHY, A. L., PICCO, G. P., AND ROMAN, G.-C. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol. 15*, 3 (2006), 279–328.

[36] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. In *SOSP* (USA, 1997), ACM.

[37] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008), ACM.

[38] SELLIS, T. K. Multiple-query optimization. *ACM Trans. Database Syst. 13*, 1 (1988), 23–52.

[39] SHAH, M. A., SHAH, M. A., CHANDRASEKARAN, S., HELLERSTEIN, J. M., HELLERSTEIN, J. M., CH, S., CH, S., FRANKLIN, M. J., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *In ICDE* (2002), pp. 25–36.

[40] TWITTER. The hadoop of real-time processing. `http://tech.backtype.com/preview-of-storm-the-hadoop-of-realtime-proce`. Accessed March, 2012.

[41] TWITTER. Twitter, 2010. `http://www.twitter.com`. Accessed May 7, 2010.

[42] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP* (2001).

[43] WODEHOUSE, P. G. My Man Jeeves. `http://www.gutenberg.org/etext/8164`. Accessed May 1, 2010.

[44] YAHOO! Yahoo! Pipes, 2010. `http://pipes.yahoo.com/pipes/`. Accessed May 7, 2010.

[45] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., LFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI* (2008).

[46] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MCCAULEY, M., MA, J., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

[47] ZHENG, Y., LI, Q., CHEN, Y., AND XIE, X. Understanding mobility based on GPS data. In *ACM conference on Ubiquitous Computing (UbiComp)* (2008).

[48] ZHENG, Y., ZHANG, L., XIE, X., AND MA, W.-Y. Mining interesting locations and travel sequences from GPS trajectories. In *WWW* (2009).

[49] ZHOU, J., LARSON, P.-A., FREYTAG, J.-C., AND LEHNER, W. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD* (2007).