Fault-tolerant Stream Processing using a Distributed, Replicated File System

YongChul Kwon
University of Washington
Seattle, WA
yongchul@cs.washington.edu

Magdalena Balazinska
University of Washington
Seattle, WA
magda@cs.washington.edu

Albert Greenberg
Microsoft Research
Redmond, WA
albert@microsoft.com

ABSTRACT

We present SGuard, a new fault-tolerance technique for distributed stream processing engines (SPEs) running in clusters of commodity servers. SGuard is less disruptive to normal stream processing and leaves more resources available for normal stream processing than previous proposals. Like several previous schemes, SGuard is based on rollback recovery [18]: it checkpoints the state of stream processing nodes periodically and restarts failed nodes from their most recent checkpoints. In contrast to previous proposals, however, SGuard performs checkpoints asynchronously: *i.e.*, operators continue processing streams during the checkpoint thus reducing the potential disruption due to the checkpointing activity. Additionally, SGuard saves the checkpointed state into a new type of distributed and replicated file system (DFS) such as GFS [22] or HDFS [9], leaving more memory resources available for normal stream processing. To manage resource contention due to simultaneous checkpoints by different SPE nodes, SGuard adds a scheduler to the DFS. This scheduler coordinates large batches of write requests in a manner that reduces individual checkpoint times while maintaining good overall resource utilization. We demonstrate the effectiveness of the approach through measurements of a prototype implementation in the Borealis [2] open-source SPE using HDFS [9] as the DFS.

1. INTRODUCTION

Today's Web and Internet services (*e.g.*, email, instant messaging, search, online games, e-commerce) must handle millions of users distributed across the world. To manage these large-scale systems, service providers need sophisticated monitoring tools: they must continuously monitor the health of their infrastructures, the quality-of-service experienced by users, and any potentially malicious activities. Stream processing engines (SPEs) [2, 48][1] are well-suited to support these types of monitoring activities as they process data streams continuously and with low latency. The scale of today's monitoring applications, however, requires running these SPEs in clusters of commodity servers.

Several distributed stream processing engines exist and could be applied to the problem [2, 11, 14]. However, an important challenge when running distributed SPEs in server clusters is fault-tolerance. As the size of the cluster grows, so does the likelihood that one or more servers will fail. Server failures include both software crashes and hardware failures. In this paper, we assume server clusters with tens of infrequently failing servers and focus only on fail-stop failures [45][2] : when a server fails (either due to a software or hardware bug), it stops processing streams, loses its in-memory state and possibly its persistent state, and loses all messages in flight to and from other servers. The failure can be detected by other servers through keep-alive messages. The server may never recover. Additionally, multiple servers can fail simultaneously.

A failure can cause a distributed SPE to block or to produce erroneous results. In both cases, applications that rely on these results are negatively impacted by the failure. To address this problem and enable a distributed SPE to achieve high-availability, several fault-tolerance techniques have recently been proposed [6, 25, 26, 46]. These techniques rely on replication: the state of stream processing operators is replicated on multiple servers. In some schemes, these servers all actively process the same input streams at the same time [6, 25, 46]. In other techniques, only one primary server performs the processing while the other secondary servers passively hold the replicated state in memory until the primary fails [25, 26]. The replicated state is periodically refreshed through a process called checkpointing to ensure the backup nodes hold a recent copy of the primary's state. Although these techniques provide excellent protection from failures, they do so at relatively high cost. When all processing is replicated, the number of servers available for normal stream processing is cut at least in half. When replicas only keep snapshots of a primary's state without performing any processing, the CPU overhead is less but the cluster still loses at least half of its memory capacity (the exact overhead depends on the degree of replication). Currently, fault-tolerance in SPEs can thus only be achieved by reserving for fault-tolerance resources that could otherwise

[1] Also called data stream managers [1, 38], stream databases [16], continuous query processors [11], or event processing engines [5, 15].

[2] Even though we assume fail-stop failures, SGuard could be used in the presence of other types of failures with the addition of standard failure detection and membership management techniques [31].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

be used for normal stream processing. Additionally, fault-tolerance can be disruptive because checkpointing the SPE state requires temporarily suspending normal processing.

In this paper, we show that it is possible to make SPEs fault-tolerant at a much lower cost and in a manner that maintains the efficiency of normal SPE operations. Our approach, called SGuard, is based on three key innovations.

First, because SPEs process streams in memory, memory is a critical resource. To save memory resources, SGuard saves checkpoints to disk. Stable storage has previously not been considered for streaming systems because saving state to disk was considered too slow for streaming applications. SGuard partly overcomes this performance challenge by using a new generation of distributed and replicated file system (DFS) such as the Google File System (GFS) [22], the Hadoop Distributed File System (HDFS) [9], and KFS [29]. In contrast to earlier distributed files systems such as NFS [39] or AFS [24], these new DFSs are optimized for reads and writes of large data volumes and for append-style workloads, which exactly matches our application. They also maintain high-availability in the face of disk failures. Furthermore, these DFSs are already being used in the context of large-scale infrastructures for Internet and Web services. They are thus readily available in the environments for which we design SGuard. Our approach leverages their presence. With SGuard, each SPE node now serves as both a stream processing and a file system node. SGuard checkpoints the operator states periodically into the DFS. If the state to checkpoint is large, SGuard partitions it into smaller chunks that are written in parallel onto many nodes in the DFS. The first innovation of SGuard is thus the use of a DFS as stable storage for SPE checkpoints.

Second, even with these new DFSs, when many nodes have large states to checkpoint, they contend for disk and network bandwidth resources, slowing down individual checkpoints. To address this resource contention problem, SGuard extends the DFS master node with a new type of scheduler, called Peace. Given a queue of write requests, Peace selects the replicas for each data chunk and schedules all writes in a manner that significantly reduces the latency of individual writes, while only modestly increasing the completion time for all writes in the queue. For example, in a 17-node network spread across two racks on a LAN, if 16 nodes need to checkpoint 512MB each, we find that Peace reduces the individual checkpoint times from 45s to less than 20s, while increasing the time to complete all checkpoints only from 55s to 75s, all this with a three-fold replication level (Section 6.2, Figure 10). Hence, Peace reduces the maximum DFS throughput slightly to achieve high performance of individual write batches. Peace achieves this goal by scheduling only as many concurrent writes as there are available resources and by selecting the destination for each write in a manner that avoids resource contention. Although developed in the context of SGuard, the Peace scheduler is generally applicable to reduce the latency of individual writes in a DFS under high levels of contention.

Third, to make checkpoints transparent, SGuard provides a new memory management middleware (MMM) that makes it possible to checkpoint the state of an SPE operator while it continues executing. MMM partitions the SPE memory into application-level "pages", *i.e.* large fragments of memory allocated on the heap, and uses application-level copy-on-write to perform asynchronous checkpoints. That

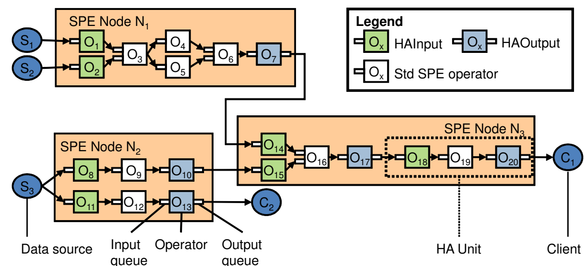

Figure 1: **SPE query diagram composed of 20 operators distributed across three nodes.**

is, to checkpoint the state of an operator, its pages are marked as read-only. They are copied to disk while the operator continues processing. The operator is only suspended when it touches a page that has not yet been recopied to disk. The system creates a copy of the page before letting the operator resume its execution. MMM checkpoints are more efficient than previous approaches. Most previous approaches for concurrent checkpointing rely on the operating system's page protection mechanism and use copy-on-write techniques to checkpoint the state of an *entire process* asynchronously [3, 23, 35] (or they emulate this approach in a virtual machine [52]). Because they must checkpoint the entire process, these techniques yield large checkpoints. In contrast, SGuard enables selective checkpoints at the application-level. Unlike other fine-grained application-level checkpointing mechanisms, SGuard offers asynchronous checkpoints and provides a good level of transparency by exposing to application writers a standard data structure interface. Finally, with the MMM, pages are written to disk without requiring any expensive translation. Overall, the MMM checkpoints are thus more efficient and less disruptive to normal stream processing than previous schemes. For example, when checkpointing 128MB of state, as soon as the state is split into four independent partitions, the MMM checkpoint latency is completely hidden, while conventional methods require more than 64 partitions to hide the latency due to checkpoints (Section 6.1, Figure 8). Even though it is designed to checkpoint the state of an SPE, the MMM is more generally applicable to efficiently checkpointing data flow applications.

In summary, SGuard's contributions are thus (1) the use of a new generation DFS such as GFS to achieve fault-tolerance in a distributed SPE, (2) the new Peace DFS scheduler, and (3) the new MMM.

2. BACKGROUND

In this section, we review the query execution model and fault-tolerance techniques used in distributed SPEs. We also present an overview of the emerging distributed, replicated file systems that SGuard uses for stable storage.

2.1 Stream Processing Engines

A large number of stream processing engines (SPEs) have recently been developed. These engines include numerous research prototypes [1, 2, 7, 11, 16, 27, 38] and several commercial products [5, 15, 48]. In an SPE, a query takes the form of a loop-free, directed graph of operators. Each operator processes data arriving on its input streams and produces data on its output stream. The processing is done

in memory without going to disk.[3] Common operators include filters (a form of selection), maps (a generalized projection), windowed aggregates, and windowed joins [1]. We assume the stream processing computation is deterministic, although rollback recovery and thus our technique can also work in face of non-determinism [18, 25].

These query graphs are called *query diagrams* and can be distributed across multiple servers in a local-area network (LAN) [2, 11, 14, 16] or a wide-area network (WAN) [2, 14]. Each server runs one instance of the SPE and we refer to it as a processing node (or simply node). Figure 1 illustrates a query diagram composed of twenty operators (O_1 through O_{20}) distributed across three nodes. In this diagram, nodes N_1 and N_2 are *upstream* from N_3 because they produce streams consumed by operators on N_3.

Several techniques have been proposed to mask failures of processing nodes and achieve high availability (HA) in distributed SPEs. Most techniques [6, 25, 46] are based on state-machine replication [45]. They achieve fault-tolerance by replicating query diagram operators across multiple nodes and having all replicas process the same input streams in parallel. With this approach, however, at least half the cluster resources are dedicated to fault-tolerance.

An alternate technique, called passive standby [25], consists in using rollback recovery [18] to achieve fault-tolerance. With this approach, only a primary replica of each node processes streams. This primary periodically takes a snapshot of its state and sends that snapshot to one or more backup nodes [25, 26] (the primary can also copy only the difference of its state since the last checkpoint). When a failure occurs, one of the backups is selected as the new primary. With this approach, the CPU overhead is smaller since backups do not need to process input streams in the absence of a failure. This approach, however, still incurs at least 100% memory overhead because checkpoints are stored in memory on the backup nodes.

In many large-scale monitoring applications, the state of stream processing queries can be large. For example, a single operator that joins two 20Mbps streams within a 10min time-window has a 3GB state. Similarly, an operator that needs to keep even a small amount of statistics (*e.g.*, 1KB) for each of a million URLs or clients, can easily end-up maintaining an aggregate state on the order of a gigabyte. Queries are composed of several such operators. Even if the state of individual operators is small or is partitioned, the aggregate state remains large.

Memory is thus a critical resource in an SPE. Our goal is to develop a fault-tolerance scheme that does not require reserving half the memory in a cluster for fault-tolerance. To achieve this goal, we propose to save checkpoints to disk. The challenge, however, is that disks have high write latencies while stream processing applications have near real-time processing requirements and can only tolerate minimal disruptions in the stream processing flow. To meet this challenge, our system uses and extends a DFS.

2.2 Distributed, Replicated File Systems

The growth of Internet-scale services requires new monitoring infrastructures but also creates unprecedented data processing needs. To address these needs, companies rely on large-scale data centers housing clusters consisting of thousands of commodity servers (which we call nodes) that store and process terabytes of data [22]. To manage such large data volumes, new types of large-scale file systems or data stores in general are emerging, including the Google File System (GFS) [22], the Hadoop Distributed File System (HDFS) [9], KFS [29], and Amazon's Dynamo [20].

[3]Some systems include operators that join streams with stored relations, but we ignore such operators in this paper.

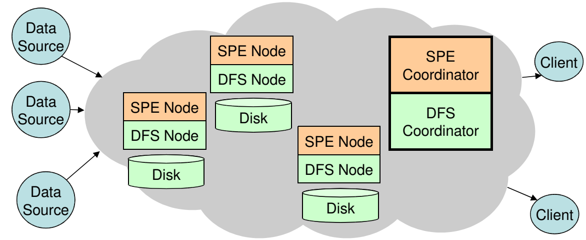

Figure 2: **SGuard system architecture.**

GFS and similar systems (KFS and HDFS) provide several features that make them especially attractive for our problem. SGuard is based on HDFS, the open-source version of GFS, which has the following two important properties:

Support for large data files. HDFS is designed to operate on large, Multi-GB files. It is optimized for append and bulk read operations. To provide high read and append performance, each file is divided into chunks. Chunks are typically large (*e.g.*, 64MB). Different chunks are stored on different nodes. A single master node (the *Coordinator*) makes all chunk placement decisions. A client that needs to read or write a file communicates with the Coordinator only to learn the location of the required chunks. All data transfers are then performed directly between the client and the individual nodes storing the data. SGuard exploits this storage infrastructure to save checkpoints of operators' states.

Fault-tolerance through replication. Because these file systems assume clusters of commodity hardware, they also assume frequent failures. To protect from failures, each data chunk is replicated on multiple nodes using synchronous master replication: the client sends data to the closest replica, which forwards it to the next closest replica in a chain, and so on. Once all replicas have the data, the client is notified that the write has completed. The level of replication is set by the client but the location of the replicas is determined by the Coordinator. The Coordinator detects node failures and creates new replicas for any under-replicated chunks. The Coordinator itself is also fault-tolerant (using rollback recovery). SGuard relies on this automatic replication to protect the distributed SPE against multiple simultaneous failures.

3. SGuard OVERVIEW

SGuard is based on passive standby, the rollback-recovery technique for SPEs mentioned in Section 2.1. Passive standby works as follows. Each SPE node takes periodic checkpoints of its state and writes these checkpoints to stable storage. SGuard uses the DFS as stable storage, while previous techniques used the memory of other SPE nodes. To take a checkpoint, a node suspends its processing and makes a copy of its state (we later discuss how SGuard avoids this suspension). For example, in Figure 1, Node N_3 takes a checkpoint by making a copy of the states of operators O_{14}

through O_{20} and their input and output queues. Between checkpoints, each node logs the output tuples it produces. In the example, operators O_7, O_{10}, O_{13}, and O_{20} log their output (as we discuss later, O_{17} also logs its output). When a failure occurs, a backup node recovers by reading the most recent checkpoint from stable storage and reprocessing all input tuples since then. For example, if N_3 fails, another node will read N_3's most recent checkpoint and will ask operators O_7 and O_{10} to replay their logged output tuples. To enable upstream nodes to remove tuples from their output queues, once a checkpoint is written to stable storage, a node sends a message to all its upstream neighbors to inform them of the checkpointed input tuples. These tuples will no longer be needed in the case of a failure and can be discarded.

Because nodes buffer output tuples, they can checkpoint their states independently and still ensure consistency in the face of a failure. Within a node, however, all interconnected operators must be suspended and checkpointed at the same time to capture a consistent snapshot of the SPE state. Such interconnected groups of operators are called HA units [26]. They are the smallest unit of work that can be checkpointed and recovered independently. SGuard also uses HA units, but it constructs them by explicitly placing new operators, that we call `HAInput` and `HAOutput`, on all input and output streams of each HA unit. As in Flux [46], these operators implement all the output tuple logging and acknowledgment logic. They enable the definition of HA units around arbitrary partitions of a query diagram. In Figure 1, there are five HA units: $\{O_1, \cdots, O_7\}$, $\{O_8, O_9, O_{10}\}$, $\{O_{11}, O_{12}, O_{13}\}$, $\{O_{14}, \cdots, O_{17}\}$, and $\{O_{18}, O_{19}, O_{20}\}$.

The level of replication of the DFS determines the level of fault-tolerance of SGuard. With $k+1$ replicas, SGuard can tolerate up to k simultaneous failures.

In SGuard, the distributed SPE and the DFS could run on two separate sets of servers. A more typical deployment, however, shares servers between these two systems as illustrated in Figure 2: each server takes the role of a data node in the DFS and a processing node in the distributed SPE. In the absence of competing workload (from applications other than the SPE), running the DFS on the same nodes as the SPE adds only a small overhead. Through micro benchmarks, not shown due to space limitation, we observed that a DFS data node uses approximately 20 to 50% of a single CPU and 100 to 150MB of RAM while receiving a chunk of data at maximum rate. When serving a chunk of data, a node uses less than 10% of a single CPU. Considering emerging multicore architectures, running DFS does not pose a significant overhead on the SPE, making it reasonable to run both on the same set of servers.

As in existing DFSs, SGuard relies on a single, fault-tolerant Coordinator to manage the placement and replication of data in the DFS. With SGuard, the Coordinator also detects SPE node failures, selects recovery nodes, and initiates recovery of lost query diagram fragments. To support these functions, all nodes periodically send keep-alive messages to the Coordinator.

A problem with the approach described so far is the impact of checkpoints on the normal stream processing flow. Because operators must be suspended during the checkpoint, each checkpoint introduces extra latency in the result stream. This latency will be exacerbated by writing checkpoints to disks rather than keeping them in memory at other nodes. To address these challenges, SGuard uses two key techniques. First, it uses a new Memory Management Middleware (MMM) illustrated in Figure 3 to (a) make checkpoints asynchronous so that operators can continue processing during checkpoints and (b) almost completely eliminate all serialization costs to speed-up the checkpointing process. Second, SGuard extends the DFS Coordinator with a scheduler, called Peace, that reduces the total time to write the state of a single HA unit while maintaining good overall resource utilization. We describe the MMM in Section 4 and Peace in Section 5.

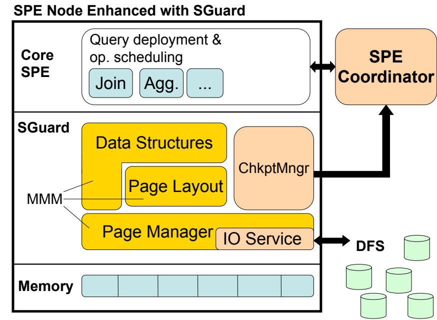

Figure 3: **SGuard software architecture.**

The SGuard technique introduces a new middleware layer for SPEs as illustrated in Figure 3 that includes the MMM. This layer also includes two other components: a *ChkptMngr* and an *IOService*. The asynchronous IOService efficiently parallelizes all reads and writes of checkpointed states to and from the DFS as we discuss further in Section 5.

The *ChkptMngr* orchestrates checkpoint and recovery operations. It checkpoints an HA unit in five steps: (1) it notifies HAInput operators that the checkpoint is starting, (2) it prepares the state of the HA unit, (3) it writes the prepared states into the DFS, (4) it informs the Coordinator about the new checkpoint, and (5) it notifies HAInput operators that the checkpoint completed. Only the first two steps are performed synchronously while the corresponding HA unit is suspended. The remaining steps proceed asynchronously while the HA unit continues execution. The first and last steps enable HAInputs to send appropriate acknowledgments to upstream HAOutputs (for output queue trimming). We omit the details of this process due to space constraints and because they are not important for SGuard's key contributions. We further present the checkpointing procedure after we describe the MMM in Section 4.

4. MMM MEMORY MANAGER

To checkpoint operators' states *without serializing them first into a buffer* (and similarly without deserializing them from a buffer upon recovery) and to enable concurrent checkpoints, where the state of an operator is copied to disk *while the operator continues executing*, SGuard must control the memory of stream processing operators. This is the role of the Memory Management Middleware (MMM).

The MMM partitions the SPE memory into a collection of "pages", which are large fragments of memory allocated on the heap. Operator states are stored inside these pages. To checkpoint the state of an operator, its pages are recopied to disk. To enable an operator to execute during the checkpoint, the MMM performs an application-level copy-

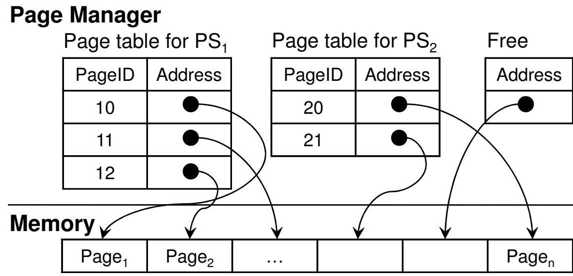

Figure 4: **Page Manager's internal data structures. Example showing two page sets PS_1 and PS_2.**

Method	Description
Page* create(PSID) void destroy(PSID)	Create a new page set Destroy a page set
Page* allocate(PSID) void release(Page*)	Allocate or release a page associated with a page set
Page* resolve(PageID)	Return the memory address of a given page

Table 1: **Page Manager's standard API.**

on-write: when the checkpoint begins, the operator is briefly suspended and all its pages are marked as read-only. The operator execution is then resumed and pages are written to disk in the background. If the operator touches a page that has not yet been written to disk, it is again briefly interrupted while the MMM makes a copy of the page.

To avoid serialization costs, the page format is identical in memory and on disk. For this, page content is constrained to sets of flat and independent records (no pointers). To facilitate application development on top of this page abstraction, the MMM exposes standard data structures such as lists, maps, and queues. Operator writers use these MMM data structures instead of ordinary data structures to store the state of operators.

Figure 3 shows the architecture of the MMM, which comprises two layers: a Page Manager (PM) layer and a Data Structures (DS) layer. The PM allocates, frees, and checkpoints pages. The DS implements data structure abstractions on top of the PM's page abstraction. To make the MMM easily extensible, we also provide a library of data structure wrappers on top of *each* page, called the Page Layout (PL) library. The PL simplifies the implementation of new data structures on top of the PM by simplifying the management of records within a page. We now present each of these components in more detail.

4.1 Page Manager Layer

As we mentioned above, the PM partitions the heap space into application-level pages. It maintains the list of free and allocated pages and controls all requests for allocating and freeing these pages. The PM also enables higher layers to group allocated pages into sets. Each set is uniquely identified with a page set identifier (`PSID`) and the PM keeps track of the pages allocated for each PSID. Figure 4 illustrates the internal data structures of the PM. Each page has a unique PageID, which is the concatenation of the PSID and a sequence number unique within the set identified by the PSID. The PM maintains and exposes a page table that maps PageIDs onto the addresses of the pages in memory. The PM manages pages only as sets. Creating meaningful relationships between pages is left to the DS layer. The DS layer associates a different PSID with each data structure instance that it creates.

Method	Description
static ADT* create(PSID,Schema)	Create a data structure with a given schema (or recover data structure from a checkpoint).
void destroy()	Destroy a data structure
insert, find, ...	Data structure specific methods

Table 2: **Data Structures' API.** This API is provided by each data structure class in the DS layer. ADT refers to a generic data structure.

Table 1 shows the PM's API. We present additional methods for checkpoint and recovery in Section 4.4. The DS uses this API to create data structure instances with unique IDs and add or remove pages from them as they grow and shrink.

4.2 Page Layout Library

The Page Layout library simplifies the implementations of data structures on top of the page abstraction by providing a seamless mechanism for operating on records within a page. Our implementation currently includes a List, Map, Queue, Dequeue, and Raw page layouts. A page layout is a wrapper for a page and has two key features: (1) it provides a data structure abstraction on top of each page, (2) it provides a level of indirection between the data structure and the underlying pages, enabling copy-on-write of the pages during checkpoints. Each page layout includes a page header region that contains at least the PageID. For example, a list page layout enables accessing the records within a page using a list API and includes the PageIDs of the preceding and following pages in the header.

4.3 Data Structures Layer

The DS layer is a collection of classes, each implementing a data structure type on top of the page abstraction exported by the PM. Each data structure operates on records with a fixed schema and without pointers to avoid serialization at the record level. Pages are written to disk directly as they appear in memory. The only deserialization that must be done during recovery is mapping PageIDs to addresses of entire pages in memory. In our current prototype, we provide list, ordered map, and unordered map data structures with similar APIs to their STL [47] counterparts. These are the main data structures required for stream processing operators, but additional data structures could be implemented.

Table 2 summarizes the API of each data structure provided by the DS. Each data structure must implement only two methods: a static method to create a new instance given a record schema (`create(PSID, Record Schema)`) and a method to destroy an instance (`destroy`). Otherwise, each data structure is free to expose any additional methods to insert, remove, or search for data items in the data structure.

Since each data structure is implemented on top of a set of pages, the data structure must somehow link these pages together for efficient data access and retrieval. For this, the data structure writes appropriate PageIDs in each page header. These PageIDs are *links* to other pages. Managing such linkage structure is specific to each data structure. Table 3 describes the content of the Super Page, which is the first page allocated for each PSID, and the content of the page headers for the three abstract data types (ADTs)

Abstract Type	Internal Representation	Super Page	Page Header
List	Doubly linked list	page id of head and tail pages	page id of previous and next pages
Ordered Map	T-tree [33]	page id of root page	page id of parent and left, right child pages
Unordered Map	Linear hash [36]	Number of directory pages and list of page ids in order	page id of previous and next pages in each bucket

Table 3: **Data structure examples.**

Method	Description
bool exists(PSID)	Returns true if a page set exists
Page* recover(PSID)	Recover an existing page set
Page* clone(Page*)	Make a copy of a page. Replace old page with new one in page set.
void chkpt(name, list<PSID>)	Checkpoint a list of page sets under a single name
void load(name)	Load all pages from a checkpoint

Table 4: **Page Manager's checkpoint and recovery API.**

implemented in our prototype.

When traversing pages, a data structure must continuously invoke the `resolve` method of the PM, but this call is fast as it only involves one hash-table lookup. As a straightforward optimization, however, the data structure can cache the mapping from PageID to in-memory address and avoid invoking the PM. As a further optimization, a data structure class can build an in-memory representation of each data structure instance with direct access to all pages in memory. For example, a list data structure can keep an in-memory list where each element of the list points to the underlying page. This redundant structure must be updated when the underlying data structure changes, but adding and removing pages is a much less frequent operation than adding and removing records. This in-memory data structure is also lost when a processing node fails and must be re-built when recovering from a checkpoint. This optimization can thus be seen as analogous to a very coarse-grained deserialization. Importantly, however, because this data structure is redundant, it can be rebuilt lazily.

4.4 Checkpoint and Recovery

By standardizing the state representation into pages, the checkpoint process becomes universal for all stream operators. It can be performed by the ChkptMngr without the participation of the operators and concurrently with the operator execution. As we mentioned above, each checkpoint is performed in five steps, but the three key steps are: (2) prepare the state of the HA unit, (3) write the prepared states into the DFS, and (4) inform the Coordinator about the new checkpoint. In the second step, the ChkptMngr prepares the state of an HA unit by calling the `chkpt` method of the Page Manager. This method is part of the PM's checkpoint/recovery API shown in Table 4. The `chkpt` method accepts two parameters: an identifier for the checkpoint and, optionally, a list of PSIDs to checkpoint. Without the list, all pages are checkpointed.

The Page Manager prepares the state of each data structure by taking a snapshot of the corresponding set of pages. To do so, the PM makes a copy of the set of all PageIDs associated with the data structure and marks each page as read-only. This process is significantly faster than serialization (as we show in Figure 6). The set of all PageIDs is then copied into a "checkpoint request". During this short state preparation operation, the HA unit must be suspended to ensure that a consistent state is checkpointed.

Once its pages are marked as read-only, an operator can resume execution, while the pages are written to disk asynchronously. After a page is written to disk, it is marked read-writable again. If an operator modifies a page while it is still marked read-only, the underlying data structure does a copy-on-write using the `clone` method of the Page Manager. The new page is read-writable so subsequent updates to the same page incur no copy overhead. The PM puts the old read-only page into a separate queue of pages that are garbage collected once written to disk.

As part of the third step of the checkpoint process, the PM hands over the checkpoint request to the underlying IOService. The latter writes to the DFS all pages identified in the checkpoint request. As soon as a page is written to disk, the IOService marks this page as read-writable again.

When all the writes complete, the IOService notifies the PM. The PM, in turn, notifies the ChkptMngr and the ChkptMngr notifies the Coordinator about the new checkpoint. As part of the notification, the ChkptMngr transmits to the Coordinator all the meta-data associated with the checkpoint, which includes the identifier of the checkpointed HA unit and the identifier of the checkpoint.

Upon a failure, the Coordinator selects one recovery node for each HA unit and informs the node of the most recent checkpoint. To restart from a checkpoint, the ChkptMngr loads the checkpoint data into the PM by invoking the PM's `load` method. Under the covers, the PM invokes the IOService to read the data pages and populate a set of free pages. As the IOService reads the pages in, the PM reconstructs the page table information on the fly for each page set. Since the page table is not ordered, it can be rebuilt independently of the order in which pages are read from disk.

Operators do not know they are being restarted from a checkpoint. They call their usual `create` method for each one of their data structures. To support recovery, the underlying data structures do not directly call the PM's `create` method. Instead, they first check if the requested instance already exists by using the `exists` method of the PM. If the instance exists, the data structures invoke the `recover` method to retrieve the appropriate Super Page.

Only HAInputs need to know that the HA unit has been recovered from a checkpoint. When it is started, each HAInput operator checks if its internal state contains information about any previously received input tuples. If it does, the HAInput operator requests that the upstream HAOutput operator replay all tuples since the last checkpoint.

If a failure affects stored data, the DFS also needs to recopy all lost chunks to ensure that the system regains its original level of replication. With SGuard, this steps is not strictly necessary since the original level of replication is automatically regained at the next checkpoint.

5. Peace SCHEDULER

To write the state of a checkpointed HA unit to the DFS, the PM uses an asynchronous IOService (Figure 3), which

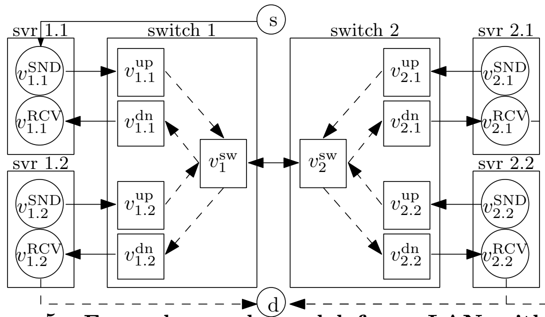

Figure 5: **Example graph model for a LAN with two switches and four servers.** Solid edges have cost 1 and dashed edges have cost 0. Capacities are not represented.

parallelizes all I/O operations to achieve good performance: the pages of an HA unit are split across multiple worker threads that write them to the DFS in parallel. The DFS further spreads the data written by each thread across many nodes. To recover the state of an HA unit, the IOService similarly reads that state from many nodes in parallel.

Checkpoints are managed under the DFS namespace. Each checkpoint is in a directory called `/checkpoint/chkpt_id`, where the chkpt_id is the concatenation of the HAUnitID and the checkpoint sequence number. Inside this directory, each worker thread writes its set of pages into one file under a unique name (*e.g.*, the thread identifier). Each file comprises one or more DFS chunks.

Within a node, the ChkptMngr can checkpoint one HA unit at a time and the above approach already provides good performance as we show in Section 6.2. However, when many nodes try to checkpoint HA units at the same time, they contend for network and disk resources, slowing down individual checkpoints. This contention is exacerbated by the replication inside the file system.

Peace addresses this resource contention problem. Peace runs inside the Coordinator at the DFS layer. Given a set of requests to write one or more data chunks, Peace schedules the writes in a manner that reduces the time to write *each set of chunks* while keeping the total time for completing all writes small. It does so by scheduling only as many concurrent writes as there are available resources, scheduling all writes from the same set close together, and by selecting the destination for each write in a manner that avoids resource contention. The advantage of adding a scheduler at the DFS rather than the application layer is that the scheduler can better control and allocate resources and we get a general, rather than application-specific, approach.

Peace takes as input a model of the network in the form of a directed graph $G = (V, E)$, where V is the set of vertices and E the set of edges. This model defines the network topology and the capacities of different resources. Figure 5 shows an example. Each server i is represented with two vertices, v_i^{SND} and v_i^{RCV}, which represent the sending and receiving processes running on server i. If a server i is connected to a switch u, then v_i^{SND} is connected to v_u^{sw} via a vertex v_i^{up}. The edge between v_i^{SND} and v_i^{up} represents the physical connection between i and u and has cost 1. An edge between v_i^{up} and v_u^{sw} represents the internal switch connection and has cost 0. The downstream connection is modeled similarly with an intermediate vertex v_i^{dn}. Vertices representing physically connected switches are also connected in the graph. In addition to a cost, each edge also has a capacity indicating the maximum number of data chunks that can be transferred along the edge in a scheduling timestep.

Algorithm 1 Peace DFS chunk scheduling algorithm.

Input: $W \Leftarrow$ a set of triples (w, r, k) where w is the writer node, r is the desired replication level, and k is the number of chunks to be written
$G \Leftarrow$ residual network graph for current timestep
$P \Leftarrow$ I/O schedule plan for current timestep.
$C \Leftarrow$ constraint templates for each replication level (*e.g.*, same node, same rack, different rack).

1: $P \leftarrow \phi$ // Initialize I/O schedule
2: $W' \leftarrow \phi$ // Unscheduled write requests
3: // For each request
4: **for all** $(w, r, k) \in W$ **do**
5: // For each chunk in the request
6: **while** $k > 0$ **do**
7: $r' \leftarrow 0$ // Current replication level
8: $z \leftarrow w$ // Current source node
9: $p \leftarrow [w]$ // Total path
10: $G' \leftarrow G$ // Temporary residual graph
11: // For each replica of the chunk
12: **while** $r' < r$ **do**
13: $H \leftarrow$ *create_graph*(G', z) // Add extra s and d nodes
14: $c \leftarrow$ *create_constraint*(C, p, r')
15: $q \leftarrow$ *randomized_shortest_path*(H, c)
16: **if** there is a path q **then**
17: $G' \leftarrow$ *residual_graph*$(G', q, 1)$ // Compute residual graph after augmenting path with one unit flow
18: $r' \leftarrow r' + 1$
19: $z \leftarrow$ last element of q
20: $p \leftarrow$ concat(p, [z])
21: **else**
22: $W' \leftarrow W' \cup \{(w, r, k)\}$
23: continue with next (w, r, k)
24: **end if**
25: **end while**
26: $k \leftarrow k - 1$
27: *P.append*(p)
28: $G \leftarrow G'$
29: **end while**
30: **end for**
31: **return** (P, G, W')

The administrator must specify the network model. Describing the network topology is simple, but determining edge capacities requires running benchmarks. This process, however, could be automated. We discuss edge capacities in Section 6.2. Because the network model specifies the cost for all resources (disk and network bandwidth), it is applicable to settings where either resource is the bottleneck.

Given the above model, Peace schedules all write requests using Algorithm 1. The basic idea is the following. Nodes submit write requests to the Coordinator, in the form of triples (w, r, k), where w is the node identifier, r is the desired replication level (typically a system-wide parameter), and k is the number of chunks to write. The algorithm iterates over all requests. For each replica of each chunk, it selects the best destination node by solving an instance of the min-cost max-flow problem over the graph G. To do so, the graph is extended with an extra source node s and destination node d. s is connected to the writer node i with an edge (s, v_i^{SND}) with capacity 1. d is connected to all servers $j \neq i$ with edges (v_j^{RCV}, d) and infinite capacity. Figure 5 shows the extended graph when SVR1.1 is the writer. The algorithm finds the minimum cost maximum flow path from s to d using a variant of the successive shortest path(SSP) algorithm [4]. The original SSP repeatedly chooses the shortest path from s to d and sends as much flow as possible along that path. In our case, because the capacity on the outgoing edge from s is 1, the algorithm reduces to finding the shortest path from s to d.

To ensure that different chunks and different replicas are written to different nodes, Algorithm 1 selects the destination for one chunk replica at a time. Additionally, it uses a *randomized* shortest path algorithm (RSP), where next hop links out of a node are always traversed in a random order, to ensure that different equivalent destinations are selected on different iterations of the algorithm. After each iteration, the algorithm computes the residual graph G'. The next iteration uses this updated graph.

So far, we described how Peace schedules writes for a single timestep. The duration of a timestep is approximately the time it takes to write one data chunk to k replicas. If only a subset of the requested chunks can be written in the next timestep, Peace runs the algorithm again on timestep $t+1$, and so on until all chunks are scheduled. For a chunk to be scheduled during a timestep, all its replicas must be written in that timestep. Because write requests arrive at the Coordinator continuously, this overall scheduling algorithm is an online algorithm. For each timestep, the algorithm has time complexity linear in the number of data chunks to schedule. The most expensive operation is randomized shortest path, which is executed once per data chunk replica.

Peace's schedule must satisfy two important properties of the DFS. First, to exploit network bandwidth resources efficiently, a node does not send a chunk directly to all replicas. Instead, it only sends it to one replica, which then transmits the data to others in a pipeline. A write is considered completed only after all replicas receive the data. To create such a pipeline, at the end of each iteration of the inner loop of Algorithm 1, the newly chosen destination node for a replica becomes the next source z, such that the series of selected destinations for the r replicas of a chunk forms a path. Second, to reduce correlations between failures, the DFS places replicas at different locations in the network (*e.g.*, one copy on the same rack and another on a different rack [9]). To achieve this property, the destination selected at each iteration must satisfy a specific set of constraints. In Peace, these constraints are specified in the form of a template. A constraint predicate c is then dynamically instantiated from the constraint templates at the beginning of each iteration. The randomized shortest path algorithm returns the shortest path that satisfies the given constraints.

To illustrate the approach, consider the graph from Figure 5. We assume that all edges have capacity 1 and that both SVR1.1 and SVR2.1 request to write one chunk with a replication factor of 2. Peace first processes the request from SVR1.1 and, using RSP, assigns the first replica to SVR1.2. It assigns the second replica to SVR2.2 because the second replica is constrained to be on a different rack. Peace now processes the request from SVR2.1. RSP cannot find any path for the first replica because the link $(v_2^{\mathrm{sw}}, v_{2.2}^{\mathrm{dn}})$ is full and the constraint requires that the first replica remain on the same rack. Thus, the request from SVR2.1 will be scheduled at $t+1$.

The above example demonstrates that our algorithm does not guarantee that it will find an optimal solution but, as we show in Section 6.2, it still enables nodes to write their individual checkpoints faster in practice.

Given the above algorithm, the file-system write protocol now works as follows. Each node submits write requests to the Coordinator indicating the number of chunks it needs to write. Peace schedules the chunks. The Coordinator then uses callbacks to let the client know when and where to write each chunk. To keep track of the progress of writes, each node informs the Coordinator every time a write completes. Once a fraction of all scheduled writes complete, Peace declares the end of a timestep and moves on to the next timestep. If the schedule does not allow a client to start writing right away, the Coordinator returns an error message. In SGuard, upon receiving such a message, the IOService propagates the message to the PM, which cancels the checkpoint by marking all pages of the HA unit read-writable again. The state of the HA unit is checkpointed again when the node can finally start writing. If the number of pages in the checkpoint changed, the node re-submits a new request instead.

Window size	256KB	512KB	1MB
Original SPE	592.1(10.5)	1144(4.84)	2109(11.1)
MMM with 8KB page	605.0(3.11)	1175(5.90)	2158(11.5)
MMM with 64KB page	604.8(3.40)	1179(4.67)	2161(9.84)

Table 5: **Per-tuple processing time and standard deviations of the Join operator in microseconds.** Average of 100 executions of processing 12K tuples.

6. EVALUATION

In this section, we evaluate SGuard by measuring the performance of its components. In Section 6.1, we measure the overhead and performance of the MMM. In Section 6.2, we measure the performance of Peace. To evaluate the MMM, we implement it as a C++ library ($\approx$ 9k lines of code) and modify the Borealis distributed SPE [2] to use it. To evaluate Peace, we implement it in the Hadoop Distributed File System [9]. Overall, we added about 500 lines of code to Borealis and 3k lines of code to HDFS.

6.1 MMM Evaluation

To evaluate the MMM, we measure its runtime overhead and its checkpoint and recovery performances.

6.1.1 Runtime Operator Overhead

We first measure the overhead of using the MMM to hold the state of an operator compared with using a standard data structures library such as STL. We study Join and Aggregate, the two most common stateful operators. In both cases, we compare the original Borealis operator implementation to our modified operator that uses the MMM. All experiments are executed on a dual 2.66GHz quad-core machine with 16GB RAM running 32bit Linux kernel 2.6.22 and single 500GB commodity SATA2 hard disk.

Table 5 shows the results for the Join operator. For each new input tuple, the Borealis Join scans all tuples in the window of the opposite stream to find matches. Because this Join implementation is not efficient, we run experiments only on window sizes of 256KB, 512KB, and 1MB. In this experiment, the Join does a self-join of a stream using an equality predicate on unique tuple identifiers. We measure the total time to process 12K tuples and compute the resulting average per-tuple processing latency. The figure shows the average from 100 executions. As the results show, the overhead of using the MMM for this operator is within 3% for all three window sizes. Additionally, the overhead is not significantly affected by the MMM page sizes.

Table 6 shows the performance for the Aggregate operator. In this experiment, the Aggregate groups input tuples

Aggregate	1MB	10MB	100MB
Original SPE	18.82(0.30)	24.93(0.27)	47.73(0.72)
MMM with 8KB page	21.25(0.33)	24.60(0.42)	31.54(0.30)
MMM with 64KB page	19.97(0.32)	23.18(0.42)	29.78(0.39)

Table 6: **Per-tuple processing time and standard deviations of Aggregate operator in microseconds.** Average of 100 executions of processing 1M tuples.

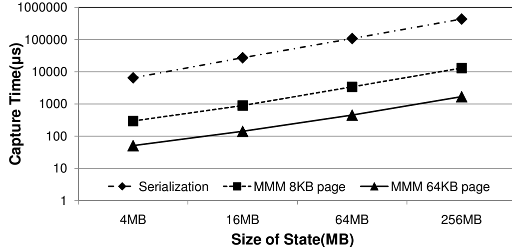

Figure 6: **Checkpoint preparation performance.** The MMM avoids the expensive state serialization cost. Average of 100 executions. Maximum standard deviation is less than 3% of the average.

by a 4-byte group name and computes count, min, and max on a timestamp field. We vary the number of groups from 1,000 to 100,000, which varies the state size from 1MB to 100MB. The results show that the MMM implementation is actually more efficient for large numbers of groups because it forces the allocation of larger memory chunks at a time, amortizing the per-group memory allocation cost.

Overall, the MMM has thus a negligible impact on operator performance. Additionally, to port Borealis operators to use the MMM required only changing 13 lines of Join operator code and 120 lines of Aggregate code[4].

6.1.2 Cost of Checkpoint Preparation

Before the state of an HA unit can be saved to disk, it must be prepared. This requires either serializing the state or copying all PageIDs and marking all pages as read-only. Figure 6 shows the time to prepare the checkpoint for a Join operator, which is faster to serialize than Aggregate because it has simpler data structures. As expected, all techniques impose an overhead linear with the size of the checkpointed state, but the MMM approach is up to two orders of magnitude faster than serialization. Avoiding serialization is thus beneficial, but the comparison is not completely fair because it ignores the copy-on-write overhead of the MMM, which we evaluate below.

6.1.3 Checkpoint Overhead

We now examine the overall runtime overhead by measuring the end-to-end stream processing latency while the SPE checkpoints the state of one operator. In this experiment, we use an Aggregate operator because it shows the worst-case performance for the MMM as it randomly rather than sequentially touches its memory pages.

The total state of the operator is 128MB, which yields 103.4MB after serialization and 143.6MB with the MMM (the extra state comes from internal fragmentation in the T-tree). We compare the performance of using the MMM

[4]Most of the change comes from the new code to efficiently handle aggregate state originally represented as nested STL types.

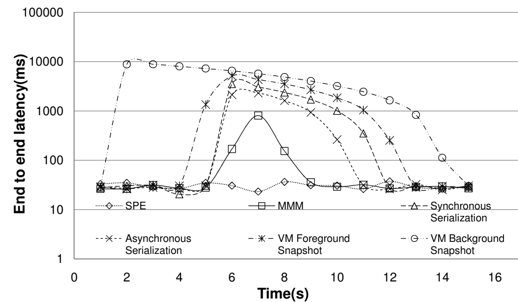

Figure 7: **End-to-end latency while taking a checkpoint of one Aggregate operator with a 128MB state size.** The overhead of the MMM is the lowest of all techniques. Over 7 consecutive checkpoints after warm up, we observed identical trends.

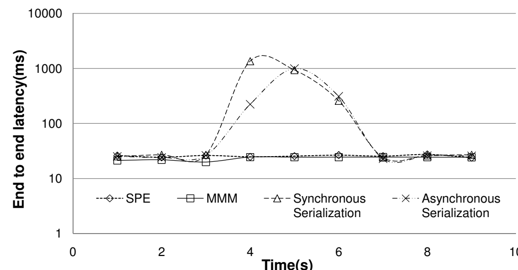

Figure 8: **End-to-end latency while taking a checkpoint of one Aggregate operator split into four.** The overhead of the MMM becomes negligible while the overhead of other techniques remains visible. Over 7 consecutive checkpoints after warm up, we observed identical trends.

against synchronously serializing state (*i.e.*, suspending the operator for the duration of both state serialization and copy to disk), and asynchronously serializing state (*i.e.*, the operator resumes execution once the state is serialized and the copy to disk proceeds in the background). We also compare with using an off-the-shelf virtual machine (VMware [52]) that can transparently checkpoint the state of the entire SPE process either while the latter is suspended (foreground snapshot) or continues executing (background snapshot). We fed 2.0K tuples/sec for 10 minutes while checkpointing every minute. Figure 7 shows the results for one checkpoint after warm up (all checkpoints displayed the same trends). All approaches cause an interruption in the normal stream processing, but the interruption due to the MMM is 2.84 times lower than that of the nearest competitor (note the log-scale on the y-axis).

A common technique for reducing checkpoint overhead is to partition the state of an operator [26, 46]. Figure 8 shows the effect of splitting the operator into four. Partitioning reduces the overhead of all techniques. The overhead of MMM becomes negligible because it reduces the time when a page is in the read-only state, thus, lowering the chance of a copy. To hide the overhead of both synchronous and asynchronous serialization, we have to split the state of the operator into 64 partitions (not shown), which adds the overhead of managing all the partitions and may not always be possible. The virtual machine cannot benefit from partitioning as it checkpoints the state of an entire process. The MMM approach is thus the least disruptive to normal stream processing.

Table 7 summarizes the average end-to-end processing latency of various techniques when we split the state across

	1 op.	4 ops	64 ops
Original SPE	23.1(1.85)	24.1(1.88)	25.0(1.81)
MMM w/o checkpointing	21.6(1.63)	21.8(1.63)	22.7(1.58)
MMM	202(645)	22.2(1.91)	22.8(1.66)
Synch. Serialization	617(1530)	157(334)	30.2(3.58)
Asynch. Serialization	302(862)	100(208)	26.6(5.49)
VM Foreground Snapshot	349(1020)		
VM Background Snapshot	299(1180)		

Table 7: **Average stream processing latency and standard deviation (in milliseconds) with various checkpointing methods for a 128MB operator state-size.** High standard deviations are due to disruptions during checkpoints when latency spikes. Overall, the MMM has the lowest overhead.

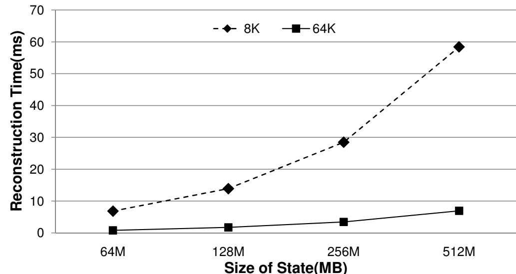

Figure 9: **Page Manager state recovery time.** Average of 100 executions. Maximum standard deviation is less than 2% of the average.

different number of operators. The results show the average of a 600 second run with 10 checkpoints per operator[5]. The overhead of the MMM is the lowest of all techniques.

6.1.4 Recovery Performance

Finally, we present the recovery performance of SGuard. Once the Coordinator detects a failure and selects a recovery node, the recovery proceeds in four phases: (1) the recovery node reads the checkpointed state from the DFS; (2) it reconstructs the Page Manager state; (3) it reconstructs any additional Data Structure state; and (4) it replays tuples logged at upstream HA units. The total recovery time is the sum of these four phases plus the failure detection and recovery node selection times. We now evaluate the performance of phases two and three. We evaluate the performance of the first phase in the next section. The last phase is the same with SGuard as with previous techniques and so we do not evaluate it again.

Figure 9 shows the performance of the Page Manager recovery. We recover the state of a 64MB to 512MB page pool using either 8KB or 64KB pages. The figure shows the average of 100 experiments. The recovery simply consists in scanning each page, associating each page with the appropriate data structure, and registering the mapping between the physical PageID and the in-memory address of the page. The time thus increases linearly with the state size and remains small overall (below 60ms for 512MB). Additionally, because pages can be processed out-of-order, this phase can proceed in parallel with reading the pages from disk. The overall overhead is thus negligible.

Recovering additional data structure information also requires scanning all pages once. This phase thus doubles the

[5]We increase checkpoint interval to 2 minutes for 64 operators not to overload the disk. Thus, 5 checkpoints per operator for 64 operators.

# of concurrent I/O	Client		Data node	
	READ	**WRITE**	**READ**	**WRITE**
1	60.5(1.10)	63.7(2.84)	42.7(3.57)	30.0(2.07)
2	56.3(3.18)	91.3(7.88)	6.16(0.35)	22.5(1.32)
4	95.3(6.15)	102(3.49)	3.04(0.31)	9.90(0.81)
8	105(9.32)	104(2.98)	1.29(0.10)	6.03(3.85)
16	110(1.78)	105(1.47)	0.63(0.03)	2.57(2.48)

Table 8: **Average and standard deviations of HDFS client throughputs (in MB/s).** Parallel I/Os at the client improve throughput as long as the network bandwidth permits. In contrast, the per-client data node performance degrades quickly with the number of concurrent I/O requests. Average of 10 executions.

overhead, which remains small overall (we do not show the numbers due to space constraints). This process, however, requires that all pages already be in memory and cannot be done in parallel with reading pages from disk. It can, however, be done lazily as the operators touch different pages.

Overall, recovery with the MMM thus imposes a low extra overhead once pages are read from disk.

6.2 IOService and Peace Evaluation

In this section, we evaluate the performance of reading checkpoints from the DFS and writing them to the DFS.

All experiments are done in a cluster of 17 machines split across two racks (8 and 9 machines each). All machines in a rack share a gigabit ethernet switch and run the HDFS data node program. The two racks are connected with a gigabit ethernet link. All machines have specifications comparable to the machine used in the previous experiments.

We first measure the performance of an HDFS client writing 1GB of data to the HDFS cluster using varying degrees of parallelism. Table 8 shows the results. As the number of threads increases, so does the total throughput at the client, until the network capacity is reached with four threads. It takes more than one thread to saturate the network because the client computes a checksum for each data chunk that it writes. In our network model, we use the more conservative capacity of 2 for the $(v_i^{\mathrm{SND}}, v_i^{\mathrm{up}})$ edges. Similarly, read performance improves with parallelism until the network capacity is reached (before running the read benchmark, we flush the OS cache). The reads are also fast. With four threads, a client reads at 95 MB/s. It could thus recover a 128MB HA unit in under two seconds.

Second, we benchmark an HDFS data node. For this, we vary the number of clients simultaneously reading a 64MB chunk from the node or writing it to the node. The table shows the read and write performances *per client*. As expected, when more clients access a data node, the performance per-client drops significantly. For reads, the aggregate performance also drops. To get good performance per-client, it is thus best if the data node serves no more than two clients at a time. In our data model, we translate this goal into a capacity of 2 on the $(v_i^{\mathrm{dn}}, v_i^{\mathrm{RCV}})$ links.

To evaluate Peace, we measure the time to write checkpoints to the DFS when multiple nodes checkpoint their states simultaneously. In this experiment, the data to checkpoint is already prepared. We only measure the I/O times. We vary the number of concurrently checkpointing nodes from 8 to 16 and the size of each checkpoint from 128MB to 512MB. The replication level is set to 3. We compare the performance against the original HDFS implementation.[6]

[6]To maximize the write throughput, we modified HDFS to

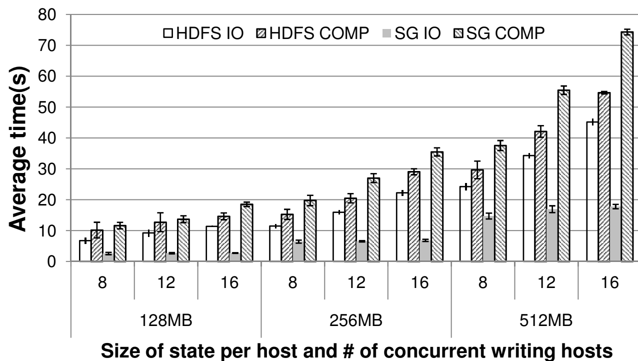

Figure 10: **Average write latencies per write request (IO) or for all writes (COMP) for varying numbers of concurrent writers with varying size I/O tasks.** Peace (SG) enables each client to complete its I/O twice as fast as HDFS, while the global completion time is only about 20% slower than in the original implementation. Average of 5 executions. We observed consistent numbers throughout the experiments.

Figure 10 shows the average time to write each checkpoint and the time to write all the checkpoints. In this experiment, Peace waits until 80% of writes complete in a timestep before starting the next one. Thus, we do not aggressively hide the latency of the schedule, resulting in an increased global completion time for larger aggregated volumes of I/O tasks. However, individual task completes their I/O much faster than in the original HDFS.

Overall, Peace thus significantly reduces the write latency for individual writes with only a small decrease in overall resource utilization. Such performance characteristics are better suited for applications such as ours where extended write latencies are undesirable (due to the copy-on-write overhead). However, because it now takes longer for all nodes to checkpoint their states, the maximum checkpoint frequency is reduced. This reduction implies somewhat longer recoveries as more tuples need to be re-processed after a failure. We believe that this is the correct trade-off since recovery with passive standby anyways imposes a small delay on streams.

7. RELATED WORK

Rollback recovery is a well known technique for making a distributed system fault-tolerant [18, 44]. Most previous approaches for concurrent checkpointing rely on the operating system's page protection mechanism and use copy-on-write techniques to checkpoint the state of an *entire process* asynchronously [3, 23, 35] (or they emulate this approach in a virtual machine [52]). As it runs entirely at the application-level, SGuard produces finer-grained checkpoints.

Several application-level checkpointing techniques exist [53]. Semi-transparent approaches [8, 42] provide APIs to specify when and what to checkpoint. Transparent approaches [8, 28] use preprocessors to produce checkpoint-capable code. SGuard is more transparent than semi-transparent approaches and more fine-gained than static or runtime analysis ones. Additionally, unlike SGuard, some of the above techniques do not provide concurrent checkpointing [8] and many require expensive state serialization [8]. The C^3 [10] application-level checkpointing system is most similar to SGuard in that it also uses a custom memory manager. However, because C^3 checkpoints the state of entire applications, its checkpoints remain more coarse-grained than those produced by the MMM. Additionally, C^3 does not support concurrent checkpoints. SRS [50] is also similar to SGuard. It provides a checkpoint framework for applications and uses a network-based storage system. However, unlike SGuard's MMM, it only checkpoints primitive data types and users must be aware of APIs to register, checkpoint, and recover application state.

SGuard's approach for writing checkpoints to disk combines ideas from striped [12] and staggered [41, 51] SGuard stripes checkpoints by writing different pieces of the checkpointed state to different DFS nodes. It staggers checkpoints by carefully assigning backup nodes and scheduling all write activities at the file system level.

The MMM shares some features with an OODBMS storage manager [17, 19, 30, 54]. Unlike an OODBMS, however, the MMM is designed to hold the state of stream processing operators. It thus only supports flat objects (no nesting and no pointers) and does not implement locking nor transactions. Furthermore, the MMM enables selective checkpoints of individual data structures.

Compared with an embedded database such as BerkeleyDB [40], the MMM exposes a greater variety of data structures that can also be individually checkpointed. Because the MMM is not a database, it is more lightweight (*e.g.*, it need not implement locking nor transactions). Finally, the MMM exposes to developers the standard data structures API, making it more transparent to use.

Peace uses the well-known min-cost max-flow [4] and first-come first-served techniques. Peace innovation lies in applying these techniques in a way that ensures: (1) different replicas are sent to different nodes that meet specific constraints, (2) data is transmitted from the source to replicas in a chain, and (3) a batch of write requests from the same source are scheduled as close together as possible.

Commercial database management systems (DBMSs) also rely on replication to provide fault-tolerance [13, 43, 49]. In contrast to a DBMS which replicates stored data, SGuard replicates the transient state of SPE queries. Also, because SPEs do not have any notion of transactions, techniques such as fuzzy checkpoints [37] do not apply. SPEs can be viewed as a kind of main-memory database [21]. However, since SPEs do not execute transactions but continuously transform streams, checkpoint and recovery techniques from main-memory databases (*e.g.*, [32, 34]) are orthogonal to fault-tolerance in SPEs.

8. CONCLUSION

We presented SGuard, a new technique for fault-tolerant distributed stream processing. SGuard leverages the existence of a new type of DFS to provide efficient fault-tolerance at a lower cost than previous proposals. Additionally, SGuard extends the DFS with Peace, a new scheduler that reduces the time to write individual checkpoints in face of high contention. SGuard also improves the transparency of SPE checkpoints through the Memory Management Middleware, which enables efficient asynchronous checkpointing. Overall, the performance of SGuard is promising. With Peace and the DFS, nodes in a 17-server cluster can checkpoint 512MB of state within less than 20s each. At the same time, the MMM efficiently hides this checkpointing activity. Many optimizations are possible to the scheme we proposed

never choose the local hard disk as a replica.

in this paper and we plan to explore them in future work.

9. ACKNOWLEDGMENTS

We thank D. Suciu, J.-H. Hwang, R. Geambasu, and the anonymous reviewers for helpful comments on drafts of this paper. This work was partially supported by NSF Grants IIS-0713123, IIS-0454425, and a gift from Cisco Systems Inc.

10. REFERENCES

[1] Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.
[2] Abadi et. al. The design of the Borealis stream processing engine. In *Proc. of the Second CIDR Conf.*, Jan. 2005.
[3] A. Agbaria and J. S. Plank. Design, implementation, and performance of checkpointing in NetSolve. In *Proc. of the Int. Conf. on Dependable Systems and Networks (FTCS-30 & DCCA-8)*, pages 49–54, June 2000.
[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications.* Prentice-Hall, Inc., 1993.
[5] Aleri. http://www.aleri.com/index.html.
[6] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 SIGMOD Conf.*, June 2005.
[7] R. Barga, J. Goldstein, M. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *Proc. of the Third CIDR Conf.*, Jan. 2007.
[8] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.
[9] D. Borthakur. The Hadoop distributed file system: Architecture and design. http://lucene.apache.org/hadoop/hdfs_design.pdf, 2007.
[10] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *In Proc. of the 9th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM.
[11] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First CIDR Conf.*, Jan. 2003.
[12] Y. S. Chang, S. Y. Cho, and B. Y. Kim. Performance evaluation of the striped checkpointing algorithm on the distributed RAID for cluster computer. In *Int. Conf. on Computational Science*, pages 955–962, 2003.
[13] Chen et. al. High availability and scalability guide for DB2 on linux, unix, and windows. IBM Redbooks http://www.redbooks.ibm.com/redbooks/pdfs/sg247363.pdf, Sept. 2007.
[14] Cherniack et. al. Scalable distributed stream processing. In *Proc. of the First CIDR Conf.*, Jan. 2003.
[15] Coral8. http://coral8.com/.
[16] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 SIGMOD Conf.*, June 2003.
[17] O. Deux. The o_2 system. *Communications of the ACM*, 34(10):34–48, Oct. 1991.
[18] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, 2002.
[19] C. et. al. Shoring up persistent applications. *SIGMOD Rec.*, 23(2):383–394, June 1994.
[20] G. DeCandia et. al. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.
[21] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE TKDE*, 04(6):509–516, 1992.
[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43. ACM Press, 2003.
[23] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. F. Bacon. Transparent recovery of Mach applications. In *Proc. of the Usenix Mach Workshop*, Oct. 1990.
[24] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.
[25] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st ICDE Conf.*, Apr. 2005.
[26] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. of the 23rd ICDE Conf.*, pages 176–185, Apr. 2007.
[27] Jain et. al. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.
[28] H. Jiang and V. Chaudhary. Migthread: Thread migration in dsm systems. In *Proc. of the 2002 Int. Conf. on Parallel Processing Workshops*, page 581, 2002.
[29] Kosmix Corp. Kosmos distributed file system(kfs). http://kosmosfs.sourceforge.net, 2007.
[30] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM*, 34(10):50–63, Oct. 1991.
[31] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, Dec. 2001.
[32] J. Lee, K. Kim, and S. K. Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proc. of the 17th ICDE Conf.*, Apr. 2001.
[33] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. of the 12th VLDB Conf.*, pages 294–303, Aug. 1986.
[34] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE TKDE*, 04(6):529–540, 1992.
[35] K. Li, J. Naughton, and J. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, 05(8), 1994.
[36] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of the 6th VLDB Conf.*, pages 212–223, Oct. 1980.
[37] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1):94–162, 1992.
[38] Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First CIDR Conf.*, Jan. 2003.
[39] Network Working Group, Sun Microsystems, Inc. RFC 1094 - NFS: Network file system protocol specification. http://www.faqs.org/rfcs/rfc1094.html, 1989.
[40] Oracle Corp. Oracle berkeley db. http://www.oracle.com/technology/products/berkeley-db/db/index.html, 2008.
[41] J. S. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, June 1993.
[42] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, Jan. 1995.
[43] A. Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. An Oracle white paper, Mar. 2002.
[44] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *Proc. of the Int. Parallel and Distributed Proc. Symp.*, Apr. 2004.
[45] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
[46] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 SIGMOD Conf.*, June 2004.
[47] Silicon Graphics, Inc. Standard template library programmer's guide. http://www.sgi.com/tech/stl/.
[48] Streambase. http://www.streambase.com/.
[49] R. Talmage. Database mirroring in SQL Server 2005. http://www.microsoft.com/technet/prodtechnol/sql/2005/dbmirror.mspx, Apr. 2005.
[50] S. S. Vadhiyar and J. Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.
[51] N. H. Vaidya. Staggered consistent checkpointing. *IEEE Trans. on Parallel and Distributed Systems*, 10(7), 1999.
[52] VMware Inc. Vmware. http://www.vmware.com, 2008.
[53] J. Walters and V. Chaudhary. Application-level checkpointing techniques for parallel programs. In *Proc. of the 3rd ICDCIT Conf.*, pages 221–234, Dec. 2006.
[54] S. J. White and D. J. Dewitt. Quickstore: a high performance mapped object store. *VLDB Journal*, 4(4):629–673, Oct. 1995.