Tsinghua Science and Technology

Volume 20 | Issue 6 Article 5

2015

Improving Performance of a Distributed File System Using a Speculative Semantics-Based Algorithm

Talluri Lakshmi Siva Rama Krishna
K L University, Andhra Pradesh, India and Research Scholar, Jawaharlal Nehru Institute of Advanced Studies （JNIAS）, Hyderabad, India.

Thirumalaisamy Ragunathan
ACE Engineering College, Hyderabad, India.

Sudheer Kumar Battula
ACE Engineering College, Hyderabad, India.

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/tsinghua-science-and-technology

Part of the Computer Sciences Commons, and the Electrical and Computer Engineering Commons

Recommended Citation

Talluri Lakshmi Siva Rama Krishna, Thirumalaisamy Ragunathan, Sudheer Kumar Battula. Improving Performance of a Distributed File System Using a Speculative Semantics-Based Algorithm. Tsinghua Science and Technology 2015, 20(6): 583-593.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Tsinghua Science and Technology by an authorized editor of Tsinghua University Press: Journals Publishing.

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/13 pp583-593
Volume 20, Number 6, December 2015

Improving Performance of a Distributed File System Using a Speculative Semantics-Based Algorithm

Talluri Lakshmi Siva Rama Krishna*, Thirumalaisamy Ragunathan, and Sudheer Kumar Battula

Abstract: File-sharing semantics is used by the file systems for sharing data among concurrent client processes in a consistent manner. Session semantics is a widely used file-sharing semantics in Distributed File Systems (DFSs). The main disadvantage of session semantics is that writes to an open file are visible to the concurrent client processes only during their next session. Recently, "linearizability semantics" was introduced in BlobSeer DFS, in which a Read Client Process (RCP) can read only a previous version of a binary large object (*blob*), while update operations are carried out on that *blob* in a concurrent manner. In this paper, we propose a new type of file-sharing semantics, namely "speculative semantics", which permits writes to an open file to be visible to other concurrent processes provided that data consistency is not affected. In addition, we propose a new read algorithm for DFSs based on speculative semantics and a new performance measurement metric called *Currency*. The experimental results obtained using BlobSeer DFS indicate that the proposed read algorithm performs better than the existing read algorithm of BlobSeer DFS.

Key words: distributed system; Hadoop; Blobseer; concurrency; speculation

1 Introduction

A Distributed System (DS) comprises a group of autonomous computers that are networked together and appear to users as a single coherent system. A Distributed File System (DFS) is one of the core components of a DS, which is used for storing and sharing data among authorized users. File-sharing in a DS is a complex and important feature.

• Talluri Lakshmi Siva Rama Krishna is with K L University, Andhra Pradesh, India and Research Scholar, Jawaharlal Nehru Institute of Advanced Studies (JNIAS), Hyderabad, India. E-mail: sivamca.mtech@gmail.com.
• Thirumalaisamy Ragunathan and Sudheer Kumar Battula are with ACE Engineering College, Hyderabad, India. E-mail: ragu_savi@yahoo.com; sudheer.itdcit@gmail.com.
∗ To whom correspondence should be addressed.
Manuscript received: 2015-01-25; revised: 2015-06-15; accepted: 2015-07-07

File-sharing semantics defines the procedure for concurrent access to shared files by read and write client programs. Many file-sharing semantics have been proposed in Ref. [1]. One of the important file-sharing semantics is POSIX, which was developed for centralized file systems. In POSIX[2] semantics, writes to a file must be immediately visible to all other processes accessing the said file. POSIX semantics is implemented in many operating systems developed for centralized environments. In a distributed environment, implementation of this semantics can cause extensive inter-node communication. Hence, it is not beneficial to use POSIX semantics for DFSs.

Andrew or session semantics is the widely used file sharing semantics in DFSs. The main disadvantage of session semantics is that writes to an open file are not immediately visible to other concurrent client processes working on same file. BlobSeer DFS abstracts data as a long sequence of bytes called a binary large object (*blob*) and creates a new version of a

blob once the execution of the *write* system call is completed. BlobSeer DFS follows linearizability semantics for sharing *blobs* among concurrent client processes. Under linearizability semantics, a read client process can read the previous version of a *blob* while a *write* system call is simultaneously executed by the write client process for updating the same *blob*.

Most web applications deployed in DSs frequently perform read operations on files and less frequently perform write operations. Hence, reading the most recently updated content from a DFS is a very important research problem, and many real-time web applications such as a "stock exchange system" require such a facility.

Speculative Processing (SP) is followed in pipeline processors for improving throughput[3]. The use of SP has also been proposed in database systems to improve the performance of transaction-processing systems[4, 5]. We used SP to improve the performance of DFSs. In this paper, we introduce a new type of file-sharing semantics, namely, "speculative semantics" for DFSs. In addition, we define a metric *Currency* based on Ref. [6], which helps determine whether the data read from a file is recent or old. Speculative semantics permits the writes to an open file to be visible to other concurrent processes under the condition that data consistency is unaffected.

In this paper, we also propose a novel read algorithm for DFSs based on speculative semantics. We assume that a read client process reads a greater number of blocks from a file, and a concurrent write client process writes only a few blocks to the file. The proposed algorithm permits a read client process (rp) to read the modifications done by only one concurrent write client process (wp), which initiated and completed its execution before rp, and the file blocks modified by wp are also read by the rp. In addition, we assume that wp modifies the file blocks only once during its execution in the system, and we allow rp to read the modifications made by wp, provided such reading does not affect data consistency. Note that such reading is not possible when following session or linearizability semantics[7].

Currently, many distributed applications are built using Hadoop and BlobSeer DFSs, which either do not permit concurrent rp and wp to access the same range of bytes of a file (rbf) or permits rp to read only the previous content of rbf, which is currently being modified by the concurrent wp. Hence, the proposed approach can be useful for improving the performance of such DFSs. For example, the stock exchange application can benefit from our approach by allowing concurrent users to read the most recently modified data from the DFS.

The remainder of this paper is organized as follows. In the next section, we describe how concurrent file accesses are carried out in various DFSs. In Section 3, we discuss session, linearizability, and speculative semantics. In Section 4, we discuss the *Currency* metric and a few applications in which currency is considered as very important. We describe the proposed algorithm in Section 5. In Section 6, we discuss our experimental results. Section 7 lists the conclusion of this study and outline of future works.

2 Related Work

In this section, we first discuss how concurrent file accesses are carried out in the Network File System (NFS), Andrew File System (AFS), and Coda File System (CFS). Next, we discuss the need for developing DFSs capable of storing, processing, and analyzing large volumes of data. Then, we describe the Google File System (GFS), Hadoop DFS (HDFS), and BlobSeer DFS. Finally, we discuss the reasons behind choosing BlobSeer DFS for implementing our proposed algorithm.

2.1 Concurrent file access in NFS

The NFS is a popular DFS developed by Sun Microsystems. An NFS server is a stateless server[1], which means that the file access information of clients is not stored in the server side main memory or disk. However, the stateless server cannot control concurrent access to its files and can therefore not guarantee file system consistency. NFS presumes that clients will not require making any concurrent modifications and does not support this facility.

2.2 Concurrent file access in AFS

AFS was developed at Carnegie Mellon University. The scalability issue[8] was overcome by assigning the majority of the work to clients rather than to the server. When a client opens a file present in AFS, a local copy of the file is stored in the client. All subsequent operations such as read and write use the local copy. In AFS, a client process (C1) performs all modifications to the local copy of the file, and the file is updated in the server only when it is closed by C1[9]. This scheme is called session semantics. In the case of concurrent

writes, the modifications made by the client who last closes the file are marked as the latest and are returned to other clients; hence, modifications made by other concurrent write clients are lost.

2.3 Concurrent file access in CFS

CFS is a descendant of AFS. In CFS, it is assumed that concurrent updates are not of prime importance. The policy of caching entire file to the local disk is followed by both AFS and CFS. Moreover, CFS uses the call back mechanism for maintaining cache coherency. The most interesting feature of CFS is the disconnected operation[10], in which files can be accessed even if the network connection has failed. To facilitate this, the file is cached locally on a disk, and a client can update the cached file even when the servers are inaccessible. CFS uses the read-one write-all method. CFS is useful for applications that require availability rather than concurrent updates.

2.4 Need for developing DFS to support data-intensive computing

With the emergence of data-intensive computing, paradigms such as Map-Reduce[11] have been developed. The Map-Reduce paradigm exploits parallelism at the data level to handle large data in a scalable manner. Early DFSs such as NFS, AFS, and CFS are unsuitable for distributed data-intensive computing. GFS, HDFS, and BlobSeer DFS were developed for processing, storing, and accessing huge amounts of data in a distributed environment. We discuss GFS, HDFS, and BlobSeer DFS in the following subsections.

2.5 Concurrent file access in GFS

GFS[12] is a proprietary DFS developed by Google in response to its own data storage needs. A centralized metadata server, called the master, is responsible for managing the directory hierarchy and the layout of each file (which chunks make up the file and where they are stored). GFS uses a relaxed consistency model. In GFS, concurrent write operations are supported, but the final result of doing so is undefined[12]. To enable concurrent write and append operations, the master employs a system of time-limited, expiring “leases”, which guarantee exclusive permission to a process to modify a chunk.

2.6 Concurrent file access in HDFS

HDFS is a distributed storage system developed under a popular open-source project called Hadoop[13]. HDFS was modeled after GFS. It was designed to be deployed on low-cost hardware; moreover, it has fault-tolerant features and provides high throughput for data-intensive applications. In HDFS, file system metadata and application data are stored in different nodes. To improve the performance of Hadoop applications, HDFS does not support POSIX standards. HDFS uses a single-writer, multiple-reader model. According to this model, at any time, only one write client process can write data to a file and many read client processes can read the same file simultaneously. In HDFS, we cannot modify a file. However, the append operation is supported in HDFS. Note that HDFS does not support concurrent write operations.

2.7 Concurrent access in BlobSeer DFS

BlobSeer DFS is designed to deal with the needs of data-intensive applications. Data is abstracted in BlobSeer DFS as long sequences of bytes called a *blob*. A *blob* comprises numerous pages, and its size can vary from 64 MB to hundreds of gigabytes. Multiple versions (snapshots) of *blobs* are available in BlobSeer DFS to support concurrent access. The BlobSeer DFS comprises architectural components as follows: (1) Data Providers (DPs), (2) Provider Manager (PM), (3) Metadata Providers (MPs), and (4) Version Manager (VM). DPs are used to store data. The PM maintains information on the available storage space in all DPs and is responsible for scheduling the placement of newly generated pages. The metadata information of BlobSeer DFS is available from MPs. The VM is responsible for assigning version numbers to *blobs*.

BlobSeer DFS is specifically optimized for an efficient fine-grained access to massive distributed data accessed under heavy concurrency[14]. BlobSeer DFS supports read and update operations on *blobs* in addition to other features such as efficient concurrent appends, concurrent writes at random offsets, and versioning[15]. Whenever a write client process updates a *blob* by invoking a *write* system call, a new version is created for that *blob* and recorded in the metadata server, provided that the recording of the metadata of all previous versions of that *blob* has already been completed. BlobSeer DFS follows linearizability semantics for sharing files among client processes, as discussed in the next section.

2.8 Reasons for selecting blobseer DFS to implement our proposed algorithm

In a previous study[16], we measured the performance of read and write operations in HDFS. The experimental results indicated that HDFS performs well with files having a size greater than the default block size and vice versa. In contrast, BlobSeer DFS[15] performs well with both small and large files. Moreover, BlobSeer DFS supports concurrent appends and concurrent writes at random offsets. Hence, we employed BlobSeer DFS for implementing our proposed algorithm.

3 Session, Linearizability, and Speculative Semantics

File-sharing semantics specifies how multiple client processes can access a shared file concurrently. In this section, we discuss session, linearizability, and speculative file-sharing semantics in detail.

3.1 Session semantics

Most DFSs have implemented Andrew or session semantics for sharing files among concurrent processes. A series of read and write accesses attempted by a client to the same file between the *open* and *close* system calls comprises a session. The main disadvantage of session semantics is that writes to an open file (f1) are not visible immediately to other concurrent client processes that have opened f1. In other words, a Read Client Process (RCP) can read modifications performed on f1 by a concurrent Write Client Process (WCP) only after that WCP closes f1. While the WCP updates f1, the RCP can only read the old contents of f1 as opposed to the modified contents of f1. The RCP can read the updated contents of f1 only during its next session, provided the WCP has already closed its session.

3.2 Linearizabiliy semantics

BlobSeer DFS follows linearizability semantics for sharing blobs among concurrent client processes. In BlobSeer DFS, whenever a WCP updates a *blob* by invoking a *write* system call, a new version of that *blob* is created and recorded in the metadata server, provided that the recording of the metadata of all previous versions of the said *blob* has already been completed. Under linearizability semantics, the RCP can read the previous version of a *blob* while a *write* system call is simultaneously executed by a concurrent WCP for updating the *blob*.

3.3 Speculative semantics

In this paper, we propose a new type of file-sharing semantics, i.e., "speculative semantics", in which writes to an open file (f2) are visible to other concurrent client processes that have opened f2. The condition here is that such speculative reading should not affect data consistency. Assume that a WCP modifies data d0 to d1 and a concurrent RCP reads d1. Now, consider that the WCP has successfully completed modifying the file system at time t1 by executing the close system call. Moreover, assume that at time t2, the RCP completes the execution of the read system call by reading d1. The RCP can complete its execution only if $t1 < t2$, else the RCP has to read only d0 and continue its execution. So, more frequently RCPs can read the recent data from the DFS by following speculative semantics.

3.4 Discussion

Under session semantics, an RCP must wait for session completion to read the modified contents of a file. Under linearizability semantics, an RCP must wait for the completion of the write system call execution on a *blob* to read the modified contents of that *blob*. Under speculative semantics, an RCP can read the modified contents of a file without waiting for the completion of the *write* system call, provided that such reading does not affect data consistency. If the reading of the modified contents of a file affects data consistency, then an RCP must read only the previous unmodified contents of the file. Hence, speculative semantics is an efficient file-sharing semantics compared with session and linearizability semantics.

4 Data Currency and Example Applications

In this section, we first discuss the metric *Currency*, which is useful for knowing whether the data read from a file is recent or old. Next, we discuss applications in which *Currency* is considered an important measure.

4.1 Data currency

Data currency expresses how up-to-date a given set of data is for a task at hand. In this study, we use the term *Currency* to denote *Data currency*. In this section, we define the metric *Currency* based on a previous study[6].

Suppose an RCP (R1) reads data object x from the storage system. The *Currency* of x provided to R1 is denoted by $c(x)$. Consider that x is created at time t3 by a WCP (W1). Now, consider that a new version of x (x_1) is created at time t5 by another WCP (W2). In

addition, R1 starts its execution at t4 such that t4 < t5, and the completion time of R1 is t6 such that t6 > t3 and t6 > t5. Then, $c(x)$ is the time difference between the creation time of the data object version x, which was read by R1, and the creation time of the most recent version of x. If the value of $c(x)$ is zero, R1 has read the most recent value. If $c(x)$ has a high positive value, R1 has read an old value. For a given case, if R1 has read x_1, the value of $c(x)$ would be zero; if R1 has read x, $c(x)$ would have high positive value. A low *Currency* value indicates high data currency, high *Currency* value indicates low data currency, and zero *Currency* value indicates the highest data currency.

Assume that data d2 was created at time t7 and the WCP modified data d2 to d3 and completed the execution of the *write* system call at time t9. Moreover, assume that the RCP concurrently started executing the *read* system call at time t8 and the execution was completed at time t10 such that t10 $\geqslant$ t9, t10 > t7, t8 > t7, and t8 < t9.

According to linearizability semantics, RCPs can only read d2 and cannot read the modified version (d3). Then, the currency value is computed as t9 − t7, which would be a high positive value; hence, the RCP will be provided with low data currency. If speculative semantics is followed in the above case, the RCP can read the recent version d3, and the computed *Currency* value is zero; such a reading does not affect data consistency.

Consider the following example that describes the computation of *Currency* under linearizability semantics. Assume that a *blob* B of version v is created at time instance 50 and the same *blob* B is modified by a WCP to produce version v1 of B with the WCP completing its execution at time instance 90. In addition, assume that at time instance 70, the RCP started reading *blob* B, and it can read only the old version v of B because version v1 of B is not available when the RCP started its execution. Next, we assume that the RCP has completed its execution of the read procedure at time instance 110. Now, the computed *Currency* value for the RCP is 40 (90–50). The disadvantage here is that the RCP cannot read the recent data, although the new version v1 is available by the time of its completion.

We consider the above case and describe how *Currency* is computed under speculative semantics. We assume that at time instance 70, the RCP starts reading *blob* B, and it can read both versions of B (v and v1) by creating additional thread. Next, we assume that the execution of both threads (main thread and additional thread) of the RCP is completed at approximately time instance 120. Given that the WCP has already completed its execution, the RCP can complete the execution of the additional thread (which has read version v1 of B) and ignore the effect of the main thread. Now, the computed *Currency* value of the RCP is 0 (90 – 90). The advantage here is that the RCP is provided with the most recent data.

4.2 Importance of data currency

The proposed speculative semantics is very useful for various applications such as stock trading and real-time systems from the viewpoint of reading recently modified data from the file system.

Consider a stock trading application that receives a market feed of stock price changes occurring during a trading day; moreover, consider that these stock prices are stored in a DFS. Assume that a particular stock price (S1) is stored in file f3 and updated at time t11 and that a client (C2) of the said stock trading application reads S1 at time t12 such that t12 > t11. If session semantics is followed by the DFS, C2 might read the updated value of S1 depending on the time of closing f3. If linearizability semantics is followed by the DFS, C2 might read the updated value of S1 depending on the time of completing the write system call for modifying S1. If speculative semantics is followed, C2 will definitely read the updated value of S1 because t12 > t11. Hence, C2 can make a better decision if speculative semantics is followed by the DFS.

In a real-time system such as a flight control system, the data read by various sensors installed in an airplane can be stored in the file system, which is entirely maintained in the main memory. In such a system, speculative semantics is useful to ensure that the flight control application can read recently modified data from the file system to make appropriate system-operation-related decisions.

5 Proposed Algorithm

In this section, we first describe the architecture of a DFS. Next, we discuss notations and assumptions used in existing and proposed algorithms. Then, we describe the existing read algorithm of the DFS. Subsequently, we describe the proposed algorithm for the DFS, which was developed based on speculative semantics. Finally, we discuss an example to show that our algorithm reads

data with high data currency from a file.

5.1 DFS architecure

We consider that a DFS has two components: Master Node (MN) and Data Node (DN). The MN is the node where the metadata of the DFS is stored, and the DN is the node where data is stored and client processes are executed. The DN is required to query the MN to obtain the addresses of DNs where data is stored.

5.2 Notations and assumptions

Notations:

- RCP: read client process;
- WCP: write client process;
- f4 and f5: files accessed by RCPs and WCPs;
- Read: read procedure for DFS;
- Write: write procedure for DFS;
- MT: main thread of Read;
- ST: speculative thread of Read;
- SBO: starting byte offset;
- EBO: ending byte offset;
- SM: shared memory;
- Readbuf: buffer used by the MT of Read for storing data read from f4;
- Writebuf: buffer used to store data modified by Write on f4;
- Stbuf: buffer used to store data read by the ST of Read from the SM of the data node where WCP is being executed;
- ubs: updated blocks.

Assumptions:

- Only one MN and one or more DNs are available in the DFS.
- Multiple RCPs and one WCP concurrently access the file f4, and at least one block is common in the range of blocks accessed by the RCPs and the WCP. The RCPs and the WCP can run in the same DN or in different DNs.
- A WCP executes the write system call only once in a session.
- Whenever a file is being modified, its status is updated in the MN.
- Existing_Read (f4, SBO, EBO) is an existing procedure specific to the DFS, and it is used for reading the range of bytes from f4.
- Main_Read_Procedure (f4, SBO, EBO, Readbuf) is the proposed read procedure in which the following procedures are executed:
  (1) Master_Read (f4, SBO, EBO, Readbuf).
  (2) Speculative_Read (f4, SBO, EBO, Stbuf).
  (3) Existing_Read (f4, SBO, EBO).
- Master_Read (f4, SBO, EBO, Readbuf) is called by the Main_Read_Procedure (f4, SBO, EBO, Readbuf) to read specified blocks of f4 from DNs.
- Speculative_Read (f4, SBO, EBO, Stbuf) is called by the Main_Read_Procedure (f4, SBO, EBO, Readbuf) to read updated blocks from main memory buffers where a WCP is being executed.
- Check_MN (f4) is the function used to verify whether f4 is being modified by querying MN. If f4 is modified, this function returns "true", else it returns "false."
- The new version of f4 is f5.
- Version_Check_MN (f4) is the function that returns the name of the latest version of file f.
- Both ST and MT are executed concurrently.
- Read_Metadata (f4, SBO, EBO) is the procedure used to obtain a list of data blocks and the DNs where these blocks are stored.
- The Write procedure is registered with the MN by providing details as follows: (1) *blob* identification number; (2) IP address of the system where the invoking WCP is being executed; and (3) SBO and EBO of the data to be modified in the *blob*.

5.3 Existing read algorithm of DFS

To read a range of bytes from a file, a client contacts MN and specifies the file name, range of bytes to be read from the file, and address of the local buffer where the data will be stored. In response, the MN first verifies whether the file is available in the file system. If the file is not available, an error message is generated. If the file is available, the MN determines the file blocks that contain the requested range of bytes. Next, the MN sends to the client the list of nearest DNs where requested data blocks are available. Thereafter, the client fetches the first data block from the nearest DN and places it in the local buffer. If multiple data blocks are to be read by the client, the relevant data blocks from various DNs can be fetched in parallel. The Existing_Read (f4, SBO, EBO, Readbuf) procedure is shown in Algorithm 1.

5.4 Proposed read algorithm

To read a range of bytes from a file, a client contacts the MN by specifying the file name, range of bytes to be read from the file, and address of the local buffer where the data will be stored. In response, first, the MN verifies whether the file is available in the file

Algorithm 1 Existing_Read (f4, SBO, EBO, Readbuf)

```
1: if f4 is not available then
2:     Send File not available error message to the client
3: else
4:     Read_Metadata (f4, SBO, EBO)
5:     for all b blocks of f4 in parallel do
6:         read b from corresponding nearest DN into Readbuf
7:     end for
8: end if
```

system. If the file is not available, an error message is generated. If the file is available, the MN calculates the blocks of the file that contain the requested range of bytes. Meanwhile, the client requests the MN to check whether the requested file is being modified. If the file is not being modified, the existing read procedure of DFS is followed. If the file is being modified, the client creates ST, and ST starts reading the modified blocks from the WCP and places them in the local buffer. Meanwhile, the MN sends to the client the list of nearest DNs where the requested data blocks are available. Thereafter, the MT of the client fetches the first data block from the nearest DN and places it in the local buffer. If multiple blocks are to be read by the client, the relevant data blocks from various DNs can fetched in parallel by the MT by spawning additional threads. Note that both ST and MT are executed in a parallel manner. The Main_Read_Procedure(f4, SBO, EBO, Readbuf) procedure is shown in Algorithm 2.

5.5 Example

We discuss an example to show the efficiency of the proposed algorithm. Consider that a file f4 already exists in the DFS. An RCP has to read a few specific blocks of f4. If f4 is being modified by a WCP, then RCP creates ST. On behalf of RCP, MT is already being executed, and both ST and MT are executed concurrently. Note that ST concurrently reads updated blocks from the buffers of WCP and stores these blocks in its local buffer. Similarly, MT reads the existing blocks of f4 from the DNs and stores these blocks in its local buffer. This read operation can be performed in parallel because the DFS stores replicated copies of the data blocks in different DNs.

If the read operation is completed by the MT of RCP, the Master_Read(f4, SBO, EBO, Readbuf) procedure queries MN for an updated copy of f4. If the updated copy of f4 (f5) is available, the MT of RCP waits for the completion of ST. Once ST is completed, the contents of Readbuf and Stbuf are combined and stored in Readbuf. Otherwise, ST is terminated. The advantage from the viewpoint of the RCP is that it can read the most recent data.

From the above example, we understand that the performance of read operation can be improved because the RCP can fetch the data directly from the main memory buffers of the WCP before a new stable version of f4 is recorded in the file system. However, the RCP must wait for successful update of metadata to complete its execution.

Algorithm 2 Main_Read_Procedure (f4, SBO, EBO, Readbuf)

```
/* This is the Main Read procedure */
1: if f4 is not available then
2:     Send File not available error message to the client
3: else
4:     Value=Check_MN(f4)
5:     if Value="true" then
6:         Execute Speculative_Read (f4, SBO, EBO, Stbuf)
           /* This is ST*/
7:         Execute Master_Read (f4, SBO, EBO, Readbuf)
           /*This is MT*/
8:         /* Both MT and ST are executed in parallel */
9:     else
10:        Execute Existing_Read (f4, SBO, EBO, Readbuf)
11:        /*This is MT*/
12:    end if
13: end if
Speculative_Read (f4, SBO, EBO, Stbuf)
1: ST is created
2: ST gets the IP address of the system where WCP is being
   executed.
3: for all ubs of f4 in parallel do
4:     read the updated block from WCP and store that block in
       Stbuf
5: end for
Master_Read (f4, SBO, EBO, Readbuf)
1: Read_Metadata (f4, SBO, EBO)
2: for all b blocks of f4 do
3:     read the block from the nearest DN into Readbuf
4: end for
5: Value=version_Check_MN(f4)
6: if Value="f5" then
7:     /*New version f5 is created */
8:     wait for the completion of ST
9:     if ST is completed then
10:        Combine the contents of Readbuf and Stbuf into
           Readbuf
11:    end if
12: else
13:    Terminate ST
14: end if
```

6 Performance Evaluation

In this section, we discuss the experimental environment employed for evaluation of the proposed algorithm, overview of our experiments, and the experimental results.

6.1 Experimental environment

Our test platform was built on a cluster consisting of eight nodes. Each node was equipped with a 2.9 GHz Intel Core 2 Duo Processor, 2 GB RAM, and a 500 GB SATA hard disk. We installed Ubuntu-12.04, kernel 3.2.1, on these nodes and installed Blobseer version 1.1 on top of it. In the eight available nodes, a metadata provider was installed in one node and a version manager and a provider manager were installed in two different nodes. The remaining five nodes were used as data providers. We modified the read procedure of BlobSeer DFS according to the requirements of our proposed algorithm.

6.2 Overview of experiments

We fixed the page size to 32 KB and considered that RCPs read an entire *blob* at once, which consisted of 1000 pages in this case. We considered the closed-queue model for submitting RCPs and WCPs to the data providers of the BlobSeer DFS. The maximum number of processes (RCPs and WCPs combined together) running at a time in the data providers was varied from 10 to 50 for performance measurement in the experiments 1, 3, 6, and 10. Moreover, we considered that at a given time, only 20% of WCPs and 80% of RCPs run in the data providers. In the second and fourth experiments, we considered that sixteen RCPs and four WCPs are executed in the system and varied the number of pages being modified from 10 to 50. Further, we considered that RCPs read the entire *blob*, which consists of 1000 pages in this case. In the first and third experiments, we fixed the number of pages being modified to 25 and varied the numbers of RCPs and WCPs from 8 to 40 (in steps of 8) and from 2 to 10 (in steps of 2), respectively.

6.3 Experimental results

Figure 1 shows the average data currency performance of RCPs in the scenario where the number of processes is varied. We can observe that the average data currency of our proposed Speculative Semantics-based Read algorithm (SSR) is better than that of the existing Linearizability Semantics-based Read algorithm (LSR) followed in BlobSeer DFS. This is because for most of the time, RCPs read the data modified by concurrent WCPs. Figure 2 shows the average data currency performance of RCPs when the number of pages modified by the WCPs is varied. Here too, SSR performs better than LSR. Figures 3 and 4 show the average read access times of RCPs. We can observe that the average read access time of the RCPs for

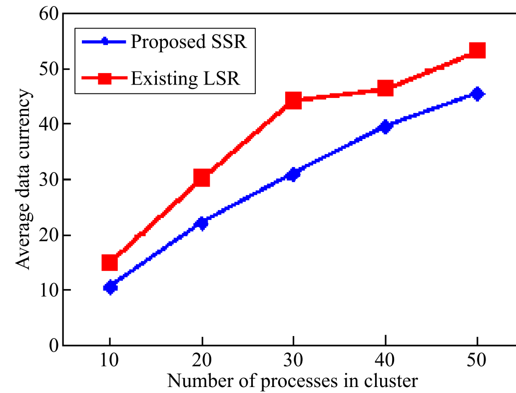

Fig. 1 Average data currency versus number of processes in the cluster.

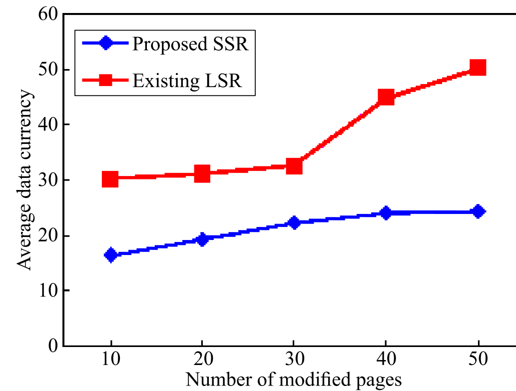

Fig. 2 Average data currency versus number of pages modified by WCPs.

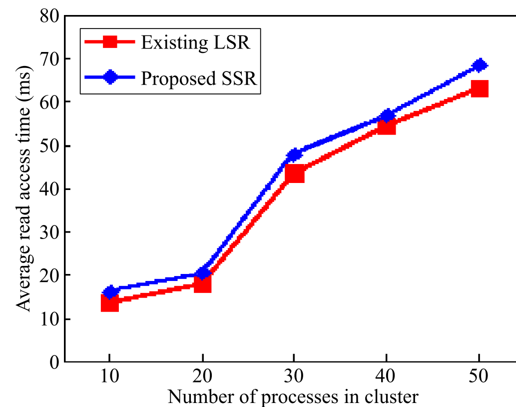

Fig. 3 Average read access time versus number of processes in the cluster.

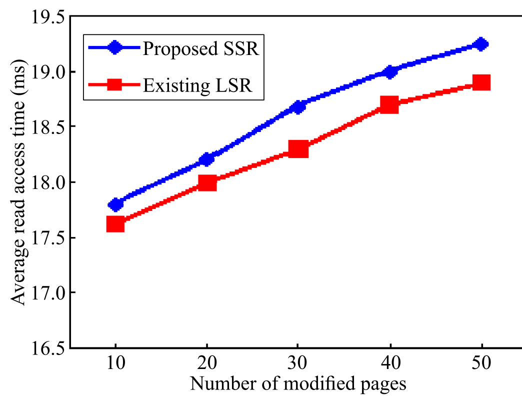

Fig. 4 Average read access time versus number of pages modified by WCPs.

the proposed SSR algorithm is slightly higher than that of the existing LSR algorithm of BlobSeer DFS. This increase in time is ascribed to the communication among that happens between RCP, WCP, and the version manager.

Figure 5 shows the number of speculative threads completed for a scenario where number of RCPs and WCPs are fixed to 16 and 4, respectively, and the number of modified pages is varied from 10 to 50. Figure 6 shows the number of speculative threads completed when the number of modified pages is fixed to 25 and the number of processes is varied from 10 to 50. We can observe from both Figs. 5 and 6 that approximately 60% of speculative threads completed which enable 60% of RCPs to read the concurrently modified data from the DFS. Figure 7 shows details pertaining to the number of speculative threads created, completed, and not completed when varying the number of RCPs and by fixing the number of modified pages to 25. We can observe that around 60% of the speculative threads complete their execution, which

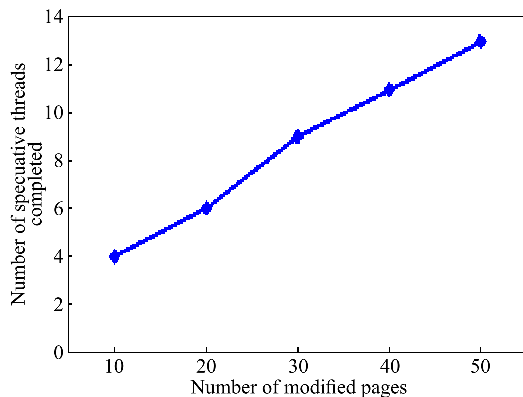

Fig. 5 Number of speculative threads completed while varying the number of modified pages.

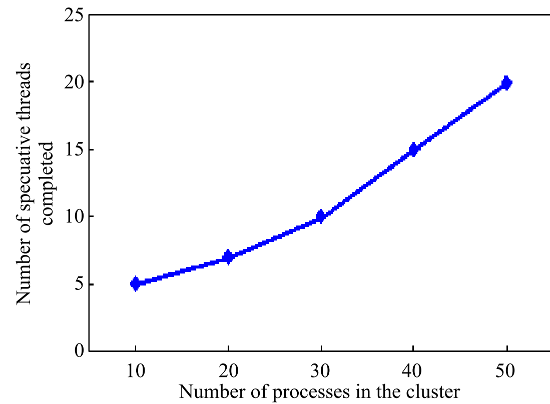

Fig. 6 Number of speculative threads completed while varying the number of processes.

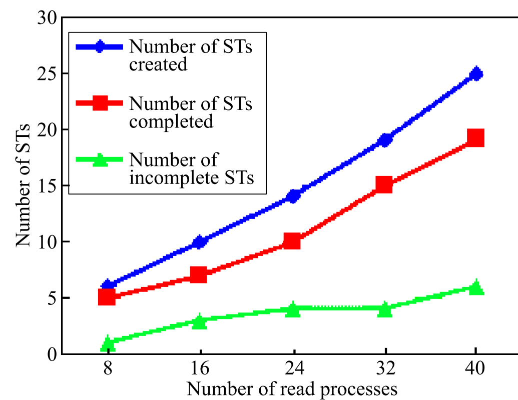

Fig. 7 Number of read processes versus number of STs.

indicates that a greater number of RCPs read the recent data written to the DFS. Figure 8 shows details pertaining to the execution time taken by the completed STs and incomplete STs. We can observe that time taken by the incomplete STs is considerably lesser than that taken by the completed STs; hence, the processing time wasted by these incomplete STs is very less. Note that modern multi-core processors have abundant computing capability. Therefore, the

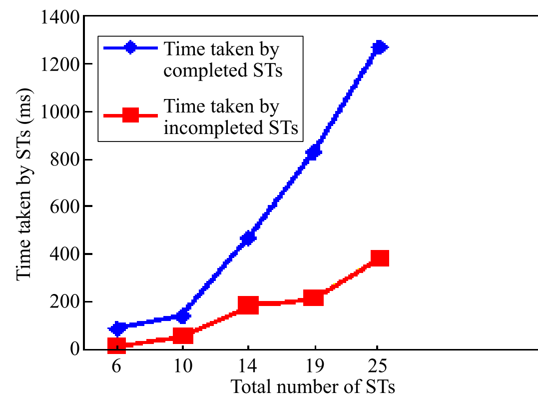

Fig. 8 Time taken by completed and incomplete STs.

processing capacity required by both the completed and the incomplete STs can be provided by computer systems built using multi-core processors.

Figure 9 depicts the registration overhead incurred for WCPs. Approximately, 0.027 ms is spent by a WCP to register its details with the MN. Figure 10 shows the time required by the RCPs to check with the MN whether a requested file is currently being modified by a concurrent WCP. We observe that approximately 0.03 ms is spent by an RCP for checking with the MN. Figure 11 shows the waiting time of the RCPs for the completion of STs to read the modifications made by the concurrent WCPs. An RCP has to wait for approximately 0.001 ms for completion of the STs. Owing to these overheads, the average block read access time of the proposed SSR algorithm is slightly higher than that of the LSR algorithm used in BlobSeer DFS.

6.4 Discussion

Our experimental study reveals that an RCP can read the new version of a *blob* as the *blob* is being modified by following our proposed SSR algorithm. In contrast, the existing LSR algorithm allows the RCP to read only the old version of the *blob* when the *blob* is being modified. Overall, we conclude that the proposed SSR algorithm performs better than the existing read algorithm of BlobSeer DFS given its ability to read the recent data written to the DFS.

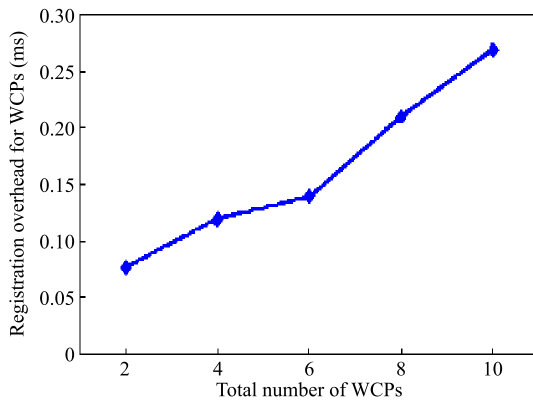

Fig. 9 Registration overhead of WCPs.

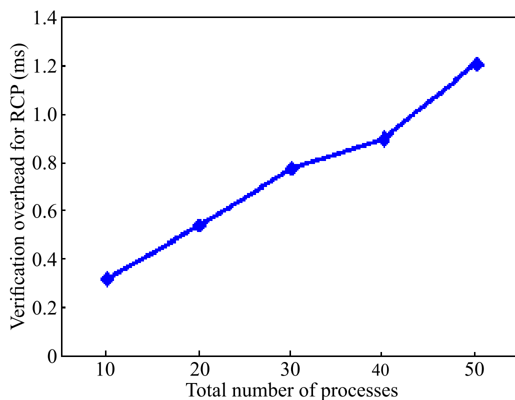

Fig. 10 Verification overhead for RCPs.

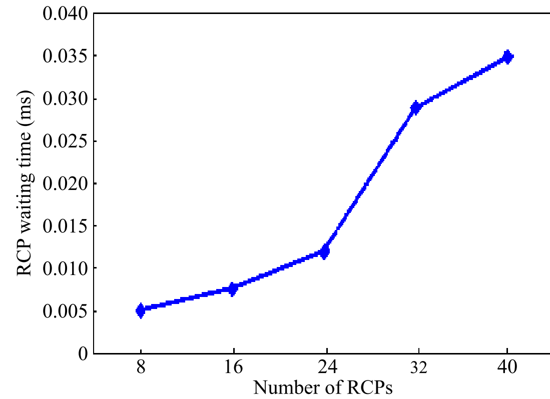

Fig. 11 RCP waiting time for the completion of STs.

7 Conclusions and Future Work

In this paper, we proposed a new type of file-sharing semantics called "Speculative semantics" and defined the metric *Currency*. In addition, we proposed a novel speculative semantics-based read algorithm for DFS. We implemented the proposed algorithm in Blobseer DFS and conducted experiments by varying the size of data and the numbers of read and write client processes in the system. The results of the experiments indicate that read client processes are able to read recent data in most of the cases without affecting data consistency. Hence, the proposed algorithm is useful for applications such as a stock exchange system, where data *Currency* is important and the availability of recent data will help users make better decisions.

The proposed algorithm permits a read client process to read the modifications made to a file by only one write client process that has started and completed its execution before the read client process, and the write client process modifies blocks of that file only once during its execution. As future work, we plan to develop an algorithm that permits a read client process to read the modifications made by multiple concurrent write client processes on a file even when the write client processes might have started their execution after the read client process and completed their execution before completion of the read client process, and the write client processes might have modified the blocks of that file more than once.

References

[1] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 2nd Ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[2] A. Josey, The single unix specification version 3, Open Group, 2004.

[3] D. Bernstein, M. Rodeh, and M. Sagiv, Proving safety of speculative load instructions at compile-time, in *ESOP'92* (pp. 56–72), Springer Berlin Heidelberg, 1992.

[4] P. Krishna Reddy and M. Kitsuregawa, Speculative locking protocols to improve performance for distributed database systems, *Knowledge and Data Engineering, IEEE Transactions on,* vol. 16, no. 2, pp. 154–169, 2004.

[5] T. Ragunathan and P. Krishna Reddy, Speculation-based protocols for improving the performance of read-only transactions, *International Journal of Computational Science and Engineering,* vol. 5, no. 3, pp. 226–242, 2010.

[6] C. Batini, C. Cappiello, C. Francalanci, and A. Maurino, Methodologies for data quality assessment and improvement, *ACM Computing Surveys (CSUR),* vol. 41, no. 3, p. 16, 2009.

[7] M. P. Herlihy and J. M. Wing, Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 12, no. 3, pp. 463–492, 1990.

[8] 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 Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, Design and implementation of the sun network filesystem, in *Proceedings of the Summer USENIX Conference*, 1985, pp. 119–130.

[10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, Coda: A highly available file system for a distributed workstation environment, *Computers, IEEE Transactions on*, vol. 39, no. 4, pp. 447–459, 1990.

[11] J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters, *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, The google file system, in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 29–43.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, The hadoop distributed file system, in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1–10.

[14] B. Nicolae, G. Antoniu, and L. Bougge, Blobseer: How to enable efficient versioning for large object storage under heavy access concurrency, in *Proceedings of the 2009 EDBT/ICDT Workshops*, ACM, 2009, p. 1825.

[15] B. Nicolae, D. Moise, G. Antoniu, L. Bouge, and M. Dorier, Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications, in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on,* 2010, p. 111.

[16] T. L. S. R. Krishna, T. Ragunathan, and S. K. Battula, Performance evaluation of read and write operations in hadoop distributed file system, in *Proc. 2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming (PAAP 2014)*, Beijing, China, 2014, pp. 110–113.

Thirumalaisamy Ragunathan received his PhD degree in computer science and engineering from IIIT, Hyderabad, India in 2010. Currently he is a professor & Dean (Research & Development) at ACE Engineering College, Hyderabad, India. His research interests include transaction processing, concurrency control, speculative processing, distributed file systems, cloud computing, and big data analysis. He is also the Honorary Director of International School of Computer Science and Information Technology, JNIAS, Hyderabad.

Sudheer Kumar Battula received his MTech degree in 2012 in software engineering from Jawaharlal Nehru Technological University (JNTU), Hyderarabad, India in 2012. At present he is working as an assistant professor in ACE Engineering College, Hyderabad, India. He carries out research activities in distributed file systems.

Talluri Lakshmi Siva Rama Krishna received his MTech degree in computer science and technology from Andhra University College of Engineering, Visakhapatnam, India in 2010. He is pursuing his PhD in computer science and engineering from Jawaharlal Nehru Technological University (JNTU), Hyderarabad. His research interests include distributed file system, speculative processing, and cloud computing. At present he is working as an associate professor at Department of Computer Science and Engineering, K L University, Greenfields, Vaddeswaram, Guntur District-522502, Andhra Pradesh, India.