Future Generation Computer Systems 86 (2018) 1118–1133

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Causal consistency algorithms for partially replicated and fully replicated systems

Ta-Yuan Hsu [a,*], Ajay D. Kshemkalyani [b], Min Shen [b]

[a] Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL 60607, USA
[b] Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA

HIGHLIGHTS

- Causal consistency gives low latency and strong semantics in replicated systems.
- We explore partial and full replication to implement causal consistency.
- We propose causal consistency algorithms for partially and fully replicated systems.
- Simulations show the efficacy of our algorithms in lowering meta-data overhead.

ARTICLE INFO

Article history:
Received 23 November 2016
Received in revised form 12 April 2017
Accepted 29 April 2017
Available online 17 May 2017

Keywords:
Causal consistency
Causality
Cloud computing
Dynamic resource provisioning
Partial replication
Full replication

ABSTRACT

Data replication is commonly used for fault-tolerance in reliable distributed systems. In large-scale systems, it additionally provides low latency. Recently, causal consistency in such systems has received much attention. However, existing works assume the data is fully replicated. This greatly simplifies the design of the algorithms to implement causal consistency. In this paper, we propose that it can be advantageous to have partial replication of data, and we propose two algorithms for achieving causal consistency in such systems where the data is only partially replicated. This work provides the first evidence that explores causal consistency for partially replicated distributed systems. We also give a special case algorithm for causal consistency in the full-replication case. We give simulation results to show the performance of our algorithms, and to present the advantage of partial replication over full replication.

1. Introduction

Reliable distributed shared services are growing in popularity since they can provide fault tolerance and direct services closer to end customers so as to lower access time and drive up customer engagement. Such services use the abstraction of distributed shared memory (DSM) along with replication of the data under the covers. With data replication, consistency of data in the face of concurrent reads and updates becomes an important problem. There exists a spectrum of consistency models in distributed shared memory systems [1]: linearizability (the strongest), sequential consistency, causal consistency, pipelined RAM, slow memory, and eventual consistency (the weakest). These consistency models represent a trade-off between cost and convenient semantics for the application programmer.

Informally speaking, causal consistency first defines a certain causality order on the read and write operations of an execution. It then requires that all the write operations that can be related by the causality order have to be seen by each application process in the order defined by the causality order. Causal consistency in DSM systems was proposed by Ahamad et al. [2], along with an implementation algorithm. Later, Baldoni et al. gave an improved implementation of causal memory [3]. Their implementation is optimal in the sense that the protocol can update the local copy as soon as possible, while respecting causal consistency. Specifically, additional delays due to the inability of Lamport's "happened before" relation [4] to map in a one-to-one way, cause–effect relations at the application level into relations at the implementation level (a phenomenon called *false causality*) are eliminated. False causality was identified by Lamport [4]. Causal consistency has also been studied by Mahajan et al. [5], Belaramani et al. [6], and Petersen et al. [7]. Lazy replication [8] is a client–server framework to provide causal consistency using vector clocks, where the size of the vector is equal to the number of replicas. A client can issue

* Corresponding author. Fax: +1 312 413 0024.
E-mail addresses: thsu4@uic.edu (T.-Y. Hsu), ajay@uic.edu (A.D. Kshemkalyani), victor.nju@gmail.com (M. Shen).

http://dx.doi.org/10.1016/j.future.2017.04.044
0167-739X/

updates and queries to any replica, and replicas exchange gossip messages to keep their data up-to-date.

Recently, consistency models have received attention in the context of cloud computing with data centers and geo-replicated storage, with product designs from industry, e.g., Google, Amazon, Microsoft, LinkedIn, and Facebook. The CAP Theorem by Brewer [9] states that for a replicated, distributed data store, it is possible to provide at most two of the three features: Consistency of replicas (C), Availability of Writes (A), and Partition tolerance (P). In the face of this theorem, most systems such as Amazon's Dynamo [10] chose to implement eventual consistency [11], which states that eventually, all copies of each data item converge to the same value. Besides the above three features, two other desirable features of large-scale distributed data stores are: low Latency and high Scalability [12]. Causal consistency is the strongest form of consistency that satisfies low Latency [12], defined as the latency less than the maximum (round-trip) wide-area delay between replicas. More recently, in the past few years, causal consistency has been studied and/or implemented by numerous researchers [12–19] in the context of geo-replicated storage. However, these implementations do not achieve optimality in the sense defined by Baldoni et al., and some of these do not provide scalability as they use a form of log serialization and exchange to implement causal consistency.

A few of the recent works on causal consistency in the geo-replicated cloud are outlined next. COPS [12] implements a causally consistent key-value store system. It computes a list of dependencies whenever an update occurs, and the update operation is not performed until updates in the dependencies are applied. The transitivity rule of the causality relationship is used to prune the size of the dependency list. Eiger [19], an improvement over COPS, provides scalable causal consistency for the complex column-family data model, as well as non-blocking algorithms for read-only and write-only transactions. The Orbe [16] key-value storage system provides two different protocols to provide causal consistency – the DM protocol uses two-dimensional matrices to track the dependencies, and the DM-Clock protocol uses loosely synchronized physical clocks to support read-only transactions. The GentleRain [17] causally consistent key-value store uses a periodic aggregation protocol to determine whether updates can be made visible in accordance with causal consistency. Rather than using explicit dependency check messages, it tracks causal consistency by attaching to updates, scalar timestamps derived from loosely synchronized physical clocks. Swiftcloud [20] provides efficient reads and writes using an occasionally stale but causally consistent client-side local cache mechanism. The size of the meta-data is proportional to the number of data centers used to store data.

All the above works, including Ahamad et al. [2] and Baldoni et al. [3], assume Complete Replication and Propagation (CRP) based protocols. These protocols assume full replication and do not consider the case of partial replication. This is primarily because full replication makes it easy to implement causal consistency. Concurrently with our work on partial replication, Crain et al. [21] outlined a causally consistent protocol for geo-distributed partial replication with dependency vectors. The main idea used is that the sender, instead of the receiver, checks the dependencies when propagating updates among data centers. However, it still considers imprecise representation of dependencies, which can result in false dependencies.

Case for partial replication

Our proposed protocols for causal consistency are designed for partial replication across the DSM system. We make a case for partial replication.

1. Partial replication is more natural for some applications. Consider the following example. A user U's data is replicated across multiple data centers located in different regions. If user U's connections are located mostly in the Chicago region and the US West coast, the majority of views of user U's data will come from these two regions. In such a case, it is an overkill to replicate user U's data in data centers outside these two regions, and partial replication has very small impact on the overall latency in this scenario.
2. With p replicas placed at some p of the total of n data centers, each write operation that would have triggered an update broadcast to the n data centers now becomes a multicast to just p of the n data centers. This is a direct savings in the number of messages and p is a tunable parameter. Thus, partial replication reduces the number of messages sent with each write operation. Although the read operation may incur additional messages to read from a remote replica if there is no local replica, the overall number of messages will still be lower than the case of full replication if the replication factor is low and the access pattern is such that readers tend to read variables from the local replica instead of remote ones. Hadoop HDFS and MapReduce is one such example. The HDFS framework usually chooses a small constant number as the replication factor even when the size of the cluster is large. Furthermore, the MapReduce framework tries its best to satisfy data locality, i.e., assigning tasks that read only from the local machine. In such a case, partial replication generates much less messages than full replication. For write-intensive workloads, it naturally follows that partial replication gives a direct savings in the number of messages without incurring any delay or latency for remote reads.
3. Partial replication allows a direct savings in resources for storage and networking hardware.
4. Recent researchers have explicitly acknowledged that providing causal consistency under partial replication is a big challenge. For example, Lloyd et al. [12] and Bailis et al. [14] write: "While weaker consistency models are often amenable to partial replication, allowing flexibility in the number of datacenters required in causally consistent replication remains an interesting aspect of future work". Although partial replication can avoid taking unnecessary network capacity and hardware resources theoretically, it is a challenge to implement partial replication compared with full replication. This is primarily because of the higher complexity and overheads (e.g., additional communication workloads and larger meta-data) of tracking causal dependency between operations.
5. The supposedly higher cost of tracking dependency meta-data, which has deterred prior researchers from considering partial replication, is relatively small for applications such as Facebook, where photos, videos and large files are also uploaded. Although light-weight data like comments, likes, and chats are also posted, it is worth exploring the size of meta-data overheads in our partial replication protocols.

Currently, there are no experimental or simulation results about the analysis of the performance trade-off between full replication and partial replication. Several algorithms that aim at achieving a causal message ordering have been previously proposed [22–24]. Different from the DSM causal consistency algorithms reviewed above, these algorithms are for message passing systems where application processes communicate with each other via sending and receiving messages. Putting aside this difference, none of these causal message ordering algorithms assume that messages get broadcast each time application processes communicate with

each other. This is similar to partially replicated DSM systems, where an individual application process writing a variable does not write to all sites in the system. In both cases, the changes in one application process do not get propagated to the entire system. These algorithms provide a good starting point for the design of our algorithms.

Contributions

We present the first algorithms for implementing causal consistency in partially replicated distributed shared memory systems.

1. Algorithm Full-Track is optimal in the sense defined by Baldoni et al. [3], viz., the protocol can update the local copy as soon as possible while respecting causal consistency. This reduces the false causality in the system.
2. Algorithm Full-Track can be made further optimal in terms of the size of the local logs maintained and the amount of control information piggybacked on the update messages, by achieving minimality. The resulting algorithm which optimally minimizes the size of meta-data is Algorithm Opt-Track.
3. As a special case of Algorithm Opt-Track, we present Algorithm Opt-Track-CRP, that is optimal in a fully replicated distributed shared memory system. This algorithm is optimal not only in the sense of Baldoni et al. but also in the amount of control information used in local logs and on update messages, which is considerably less than for algorithm Opt-Track, making it highly scalable. The algorithm is significantly more efficient than the Baldoni et al. protocol *optP* [3] for the complete replication case.

This paper provides the first evidence that explores the trade-off between partial replication and full replication analytically. We quantitatively evaluate the performance of the three protocols – Full-Track, Opt-Track, and Opt-Track-CRP – for implementing causal consistency under partial replication and under full replication. We first simulate the Opt-Track and Full-Track protocols within partially replicated systems to compare their performance. We present that Opt-Track outperforms Full-Track in network capacity and shows the advantage in write-intensive workloads. Then, we simulate Opt-Track-CRP and *optP* [3] within fully replicated systems to compare their efficiency. We present that Opt-Track-CRP also significantly outperforms *optP* in scalability and network capacity utilization.

In addition to simulating the performances within partially replicated systems and within fully replicated systems, we also explore the trade-off between partial replication and full replication analytically. We show the advantage of partial replication over full replication.

Our protocols are applicable to large-scale DSM systems, and in particular to those accommodating replications of medium or large-sized data files ($>$ 100 KB). An example of a real world network which can benefit from our results is the multimedia object oriented social network 'flickr' which is a photo-sharing social community where the average file size is 0.6 MB.

We note that recent papers, such as COPS [12], Eiger [19], Orbe [16] and GentleRain [17], implement causal consistency in large geo-replicated data storage by using a two-level hierarchical architecture (i.e., client-cluster framework) across the wide-area. They all adopt the *full-replication-centric* model, though the data is partitioned at each cluster in order to provide good scalability. In this paper, we provide the first study of *partial replication* algorithms for causal consistency, and we adopt a one-level architecture, similar to [2,3]. We intend to adapt the results of this paper to a two-level hierarchical partial geo-replication system architecture in the future.

A brief announcement of these results appears as [25] and an earlier version of these results appears as [26,27].

Organization

Section 2 gives the causal distributed shared memory model. Section 3 presents Algorithm Full-Track which implements causal consistency in the partially replicated DSM system and is optimal in the sense that it can apply updates as soon as possible. Section 4 presents Algorithm Opt-Track which implements causal consistency in the partially replicated DSM system and is additionally optimal in the sense that the size of local logs and message overheads are minimized. Section 5 presents Algorithm Opt-Track-CRP, which is a special case of Algorithm Opt-Track for fully replicated systems. Section 6 analyzes the complexity of the algorithms. Section 7 presents the communication models for simulating the optimal protocols proposed under partial replication and under full replication. Section 8 shows all the simulation results and analytically illustrates the performance trade-off between partial replication and full replication. Section 9 gives a discussion. Section 10 concludes.

2. System model

2.1. Causally consistent memory

The system model is based on that proposed by Ahamad et al. [2] and Baldoni et al. [3]. We consider a system which consists of n application processes $ap_1, ap_2, \ldots, ap_n$ interacting through a shared memory $\mathcal{Q}$ composed of q variables $x_1, \ldots, x_q$. Each ap_i can perform either a *read* or a *write* operation on any of the q variables. A *read* operation performed by ap_i on variable x_j which returns value v is denoted as $r_i(x_j)v$. Similarly, a *write* operation performed by ap_i on variable x_j which writes the value u is denoted as $w_i(x_j)u$. Each variable has an initial value $\perp$.

By performing a series of *read* and *write* operations, an application process ap_i generates a local history h_i. If a local operation o_1 precedes another local operation o_2, we say o_1 precedes o_2 under *program order*, denoted as $o_1 \prec_{po} o_2$. The set of local histories h_i from all n application processes form the global history H. Operations performed at distinct processes can also be related using the *read-from order*, denoted as $\prec_{ro}$. Two operations o_1 and o_2 from distinct processes ap_i and ap_j respectively have the relationship $o_1 \prec_{ro} o_2$ if there are variable x and value v such that $o_1 = w(x)v$ and $o_2 = r(x)v$, meaning that *read* operation o_2 retrieves the value written by the *write* operation o_1. It is shown in [2] that

- for any operation o_2, there is at most one operation o_1 such that $o_1 \prec_{ro} o_2$;
- if $o_2 = r(x)v$ for some x and there is no operation o_1 such that $o_1 \prec_{ro} o_2$, then $v = \perp$, meaning that a read with no preceding write must read the initial value.

With both the *program order* and *read-from order*, the *causality order*, denoted as $\prec_{co}$, can be defined on the set of operations O_H in a history H. The causality order is the transitive closure of the union of local histories' program order and the read-from order. Formally, for two operations o_1 and o_2 in O_H, $o_1 \prec_{co} o_2$ if and only if one of the following conditions holds:

1. $\exists ap_i$ such that $o_1 \prec_{po} o_2$ (program order)
2. $\exists ap_i, ap_j$ such that o_1 and o_2 are performed by ap_i and ap_j respectively, and $o_1 \prec_{ro} o_2$ (read-from order)
3. $\exists o_3 \in O_H$ such that $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure).

Essentially, the causality order defines a partial order on the set of operations O_H. For a shared memory to be *causal memory*, all the write operations that can be related by the causality order have

to be seen by each application process in the order defined by the causality order. More formally, we state as follows.

Given a history H, S is a *serialization* of H if S is a sequence containing exactly the operations of H such that each read operation of a variable x returns the value written by the most recent precedent write on x in S. A serialization respects a given order if, for any two operations o_1 and o_2 in S, o_1 precedes o_2 in that order implies that o_1 precedes o_2 in S. Let H_{i+w} be the history containing all the operations in h_i and all write operations of H.

Definition 1 (*Causally Consistent History*)**.** A history is causally consistent if for each application process ap_i, there is a serialization S_i of H_{i+w} that respects the causality order $\prec_{co}$.

Definition 2 (*Causal Memory*)**.** A memory is causal if it admits only causally consistent histories.

2.2. Underlying distributed communication system

The DSM abstraction and its causal consistency model is implemented by a memory consistency system (MCS) on top of the underlying distributed message passing system which also consists of n sites connected by FIFO channels. The distributed system is asynchronous. We assume that all messaging primitives are reliable. Message transfer delay is arbitrary but finite, and there is no bound on the relative process speeds. Each site s_i hosts an application process ap_i. The local MCS process at s_i is denoted mp_i.

With a partially replicated system, each site holds only a subset of variables $x_h \in \mathcal{Q}$. For application process ap_i, we denote the subset of variables kept on the site s_i as X_i. We assume the replication factor of the DSM system is p and the variables are evenly replicated on all the sites. This assumption is justified from a statistical viewpoint. It follows that the average size of X_i is $\frac{pq}{n}$.

If a variable x_h is not locally replicated, then a read operation fetches the value from one of the replica sites of x_h. The replica site could be chosen randomly from one of the sites that replicate x_h. (Alternatively, a policy decision could be made to fetch the variable from the closest replica.) For this, we assume a static system and that complete knowledge about the partial replication scheme is known to all replicas. Thus, for both a read and a write operation on a variable, each site is assumed to know the replica set of that variable.

To facilitate the read and write operations in the DSM abstraction, the underlying message passing system provides several primitives to enable the reliable communication between different sites. For the write operation, each time an application process ap_i performs $w(x_1)v$, the local MCS process invokes the `Multicast`(m) primitive to deliver the message m containing $w(x_1)v$ to all sites that replicate the variable x_1. For the read operation, there is a possibility that an application process ap_i performing read operation $r(x_2)u$ needs to read x_2's value from a remote site since x_2 is not locally replicated. In such a case, the local MCS process invokes the `RemoteFetch`(m) primitive to deliver the message m to a random site replicating x_2 to fetch its value u. This is a synchronous primitive, meaning that it will complete when the variable's value is returned. If the variable to be read is locally replicated, then the application process is simply returned the local value.

The read and write operations performed by the application processes cause *events* to be generated in the underlying message passing system. More specifically, each MCS process mp_i generates a set of events E_i. $E = \langle E_i, \ldots, E_n \rangle$ is the global set of events, ordered by Lamport's "happened before" relation $\rightarrow$ [4]. The distributed computation $\widehat{E}$ is the partial order induced on E by $\rightarrow$. The set of messages in $\widehat{E}$ is denoted $M_{\widehat{E}}$.

The following types of events are generated at each site:

- *Send event.* The invocation of `Multicast`(m) primitive by MCS process mp_i generates event $send_i(m)$.
- *Fetch event.* The invocation of `RemoteFetch`(m) primitive by MCS process mp_i generates event $fetch_i(f(x))$. Here, $f(x)$ denotes the variable being fetched.
- *Receive event.* The receipt of a message m at site s_i generates event $receive_i(m)$. The message m can correspond to either a $send_j(m)$ event or a $fetch_j(f(x))$ event.
- *Apply event.* When applying the value written by the operation $w_j(x_h)v$ to variable x_h's local replica at site s_i, an event $apply_i(w_j(x_h)v)$ is generated.
- *Remote return event.* After the occurrence of event $receive_i(m)$ corresponding to the remote read operation $r_j(x_h)u$ performed by ap_j, an event $remote\_return_i(r_j(x_h)u)$ is generated which transmits x_h's value u to site s_j.
- *Return event.* Event $return_i(x_h, v)$ corresponds to the return of x_h's value v either fetched remotely through a previous $fetch_i(f(x))$ event or read from the local replica.

To implement the causal memory in the DSM abstraction, each time an update message m corresponding to a write operation $w_j(x_h)v$ is received at site s_i, a new thread is spawned to check when to locally apply the update. The condition that the update is ready to be applied locally is called, as in [3], the activation predicate. This predicate, $A(m_{w_j(x_h)v}, e)$, is initially set to *false* and becomes *true* only when the update $m_{w_j(x_h)v}$ can be applied after the occurrence of local event e. The thread handling the local application of the update will be blocked until the activation predicate becomes *true*, at which time the thread writes value v to variable x_h's local replica. This will generate the $apply_i(w_j(x_h)v)$ event locally. Thus, the key to implement the causal memory is the activation predicate.

2.3. Activation predicate

2.3.1. The $\rightarrow_{co}$ relation

To formulate the activation predicate, Baldoni et al. [3] defined a new relation, $\rightarrow_{co}$, on *send events* in $\widehat{E}$ generated in the underlying message passing system. We modify their definition by adding condition (3) to accommodate the partial replication scenario.

Definition 3 ($\rightarrow_{co}$ *On Send Events*)**.** Let $w(x)a$ and $w(y)b$ be two write operations in O_H. Then, for their corresponding send events in the underlying message passing system, $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b})$ iff one of the following conditions holds:

1. $i = j$ and $send_i(m_{w(x)a})$ locally precedes $send_j(m_{w(y)b})$
2. $i \neq j$ and $return_j(x, a)$ locally precedes $send_j(m_{w(y)b})$
3. $i \neq j$ and $\exists l$, such that $apply_l(w(x)a)$ locally precedes $remote\_return_l(r_j(x)a)$ which precedes (as per Lamport's $\rightarrow$ relation) $return_j(x, a)$ which locally precedes $send_j(m_{w(y)b})$
4. $\exists send_k(m_{w(z)c})$, such that $send_i(m_{w(x)a}) \rightarrow_{co} send_k(m_{w(z)c}) \rightarrow_{co} send_j(m_{w(y)b})$.

Actually, it is easy to show the following property, along the lines of [3].

Property 1. $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b}) \Leftrightarrow w(x)a \prec_{co} w(y)b$.

Definition 4 (*Safety*)**.** Let $\widehat{E}$ be a distributed computation generated by a protocol P. P is safe if and only if: $\forall m_w, m_{w'} \in M_{\widehat{E}}$, $send_j(m_w) \rightarrow_{co} send_k(m_{w'})$ implies that at all common destinations s_i of the two updates, $apply_i(w)$ locally precedes $apply_i(w')$.

An activation predicate of a safe protocol has to stop the application of any update message m_w that arrives out of order with respect to $\rightarrow_{co}$.

2.3.2. Optimal activation predicate

With the $\rightarrow_{co}$ relation defined, Baldoni et al. gave an optimal activation predicate in [3] as follows:

Definition 5 (*Optimal Activation Predicate*)**.**

$$A_{OPT}(m_w, e) \equiv \nexists m_{w'} : (send_j(m_{w'}) \rightarrow_{co} send_k(m_w) \wedge apply_i(w') \notin E_i|_e)$$

where $E_i|_e$ is the set of events that happened at the site s_i up until e (excluding e).

A protocol is optimal in terms of the activation predicate if its activation predicate is false only if there exists an update message $m_{w'}$ such that $send_j(m_{w'}) \rightarrow_{co} send_k(m_w)$ and $m_{w'}$ has not yet been applied locally at s_i. The key to design the optimal activation predicate is to track dependencies under the $\rightarrow_{co}$ relation and not the Lamport's $\rightarrow$ relation.

This activation predicate cleanly represents the causal memory's requirement: a write operation shall not be seen by an application process before any causally preceding write operations. It is optimal because the moment this activation predicate $A_{OPT}(m_w, e)$ becomes true is the earliest instant that the update m_w can be applied.

3. Full-Track algorithm

3.1. Basic idea

Since the system is partially replicated, each application process performing a write operation will only write to a subset of all the sites in the system. Thus, for an application process ap_i and a site s_j, not all write operations performed by ap_i will be seen by s_j. This makes it necessary to distinguish the destinations of ap_i's write operations. The activation predicate A_{OPT} can be implemented by tracking the number of updates received that causally happened before under the $\rightarrow_{co}$ relation. In order to do so in a partially replicated scenario, it is necessary for each site s_i to track the number of write operations performed by every application process ap_j to every site s_k. We denote this value as $Write_i[j][k]$. Application processes also piggyback this clock value on every outgoing message generated by the `Multicast`(m) primitive. The *Write* matrix clock tracks the causal relation under the $\rightarrow_{co}$ relation, rather than the causal relation under the $\rightarrow$ relation.

Another implication of tracking under the $\rightarrow_{co}$ relation is that the way to merge the piggybacked clock with the local clock needs to be changed. Under the $\rightarrow_{co}$ relation, it is reading the value that was written previously by another application process that generates a causal relationship between two processes. Thus, the *Write* clock piggybacked on messages generated by the `Multicast`(m) primitives should not be merged with the local *Write* clock at the message reception or at the apply event. It should be delayed until a later read operation which reads the value that comes with the message and generates the corresponding return event.

3.2. The algorithm

With the above discussion, we now give the formal algorithm in Algorithm 1. At each site s_i, the following data structures are maintained:

1. $Write_i[1 \ldots n, 1 \ldots n]$: the *Write* clock (initially set to 0s). $Write_i[j, k] = a$ means that the number of updates sent by application process ap_j to site s_k that causally happened before under the $\rightarrow_{co}$ relation is a.
2. $Apply_i[1 \ldots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i.

Algorithm 1: Full-Track Algorithm (Code at site s_i)

WRITE(x_h, v):

1 **for** *all sites* s_j *that replicate* x_h **do**
2 $\quad Write_i[i, j] + +$;
3 `Multicast`$[m(x_h, v, Write_i)]$ to all sites s_j $(j \neq i)$ that replicate x_h;
4 **if** x_h *is locally replicated* **then**
5 $\quad x_h := v$;
6 $\quad Apply_i[i] + +$;
7 $\quad LastWriteOn_i\langle h\rangle := Write_i$;

READ(x_h):

8 **if** x_h *is not locally replicated* **then**
9 $\quad$ `RemoteFetch`$[f(x_h)]$ from any site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
10 $\quad \forall k, l \in [1 \ldots n]$, $Write_i[k, l] :=$ $\max(Write_i[k, l], LastWriteOn_j\langle h\rangle.Write[k, l])$;
11 **else**
12 $\quad \forall k, l \in [1 \ldots n]$, $Write_i[k, l] :=$ $\max(Write_i[k, l], LastWriteOn_i\langle h\rangle.Write[k, l])$;
13 return x_h;

On receiving $m(x_h, v, W)$ from site s_j:

14 **wait until** $(\forall k \neq j, Apply_i[k] \geq W[k, i] \wedge Apply_i[j] = W[j, i] - 1)$;
15 $x_h := v$;
16 $Apply_i[j] + +$;
17 $LastWriteOn_i\langle h\rangle := W$;

On receiving $f(x_h)$ from site s_j:

18 return x_h and $LastWriteOn_i\langle h\rangle$ to s_j;

3. $LastWriteOn_i\langle$variable id, $Write\rangle$: a hash map of *Write* clocks. $LastWriteOn_i\langle h\rangle$ stores the *Write* clock value associated with the last write operation on variable x_h which is locally replicated at site s_i.

4. Opt-Track algorithm

4.1. Basic idea

Algorithm Full-Track achieves optimality in terms of the activation predicate. However, in other aspects, it can still be further optimized. We notice that, each message corresponding to a write operation piggybacks an $O(n^2)$ matrix, and the same storage cost is also incurred at each site s_i. Kshemkalyani and Singhal proposed the necessary and sufficient conditions on the information for causal message ordering and designed an algorithm implementing these optimality conditions [23,24] (hereafter referred to as the KS algorithm). The KS algorithm aims at reducing the message size and storage cost for causal message ordering algorithms in message passing systems. The ideas behind the KS algorithm exploit the transitive dependency of causal deliveries of messages and encode the information being tracked.

We can adapt the KS algorithm to a partially replicated DSM system to implement causal memory there. Now, each site s_i will maintain a record of the most recent updates received from every site, that causally happened before under the $\rightarrow_{co}$ relation. Each such record also keeps a list of destinations representing the set of replicas receiving the corresponding update. When performing a write operation, the outgoing update messages will piggyback the currently stored records. When receiving an update message, A_{OPT} is used to determine when to apply the update. Once the

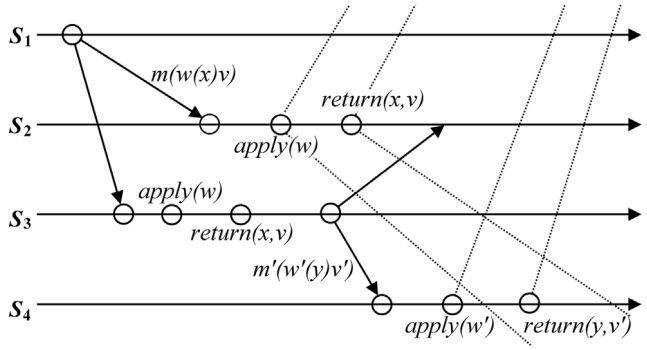


Fig. 1. Illustration of the conditions for destination information to be redundant. For causal memory algorithms, the information is "s_2 is a destination of m". The causal future of the relevant *apply* events and the causal future (under the $\rightarrow_{co}$ relation) of the relevant *return* events are shown in dotted lines.

update is applied, the piggybacked records will be associated with the updated variable. When a later read operation is performed on the updated variable, the records associated with the variable will be merged into the locally stored records to reflect the causal dependency between the read and write operations. We can prune redundant destination information using the following conditions, illustrated in Fig. 1. We use remembering *implicitly* which means inferring that information from other later or more up to date log entries, without storing that information.

- **Propagation Condition 1**: When an update m corresponding to write operation $w(x)v$ is applied at site s_2, then the information that s_2 is part of the update m's destinations no longer needs to be remembered in the causal future of the event $apply_2(w)$.
- **Pruning Condition 1**: In addition, we *implicitly* remember in the causal future (under the $\rightarrow_{co}$ relation) of event $return_2(x, v)$ (and events $remote\_return_2(r_*(x)v)$) that m has been delivered to s_2, to clean the logs at other sites.
- **Propagation Condition 2**: For two updates $m_{w(x)v}$ and $m'_{w'(y)v'}$ such that $send(m) \rightarrow_{co} send(m')$ and both updates are sent to site s_2, the information that s_2 is part of update m's destinations is irrelevant in the causal future of the event $apply(w')$ at all sites s_k receiving update m'. (In fact, it is redundant in the causal future of $send(m')$, other than m' sent to s_2.) This is because, by transitivity, applying update m' at s_2 in causal order with respect to a message m'' sent causally later to s_2 will infer the update m has already been applied at s_2.
- **Pruning Condition 2**: In addition, we *implicitly* remember in the causal future (under the $\rightarrow_{co}$ relation) of events $return_k(y, v')$ (and events $remote\_return_k(r_*(y)v')$) that m is transitively guaranteed to be delivered to s_2, to clean the logs at other sites.

The logs at the sites are cleaned as follows. The algorithm explicitly tracks (source, timestamp, Dests) per multicast message M in the log and on the message overhead. $M_{i,a}.Dests$ indicates the set of destinations of this multicast sent from source i at local timestamp a. Log entries are denoted by l and message overhead entries are denoted by o. The algorithm implicitly tracks messages that are delivered (Pruning Condition 1) or transitively guaranteed to be delivered in causal order (Pruning Condition 2) as follows.

- **Implicit Tracking 1**: $\exists d \in M_{i,a}.Dests$ such that $d \in l_{i,a}.Dests \bigwedge d \notin o_{i,a}.Dests$. Then d can be deleted from $l_{i,a}.Dests$ because it can be inferred that $M_{i,a}$ is delivered to d or is transitively guaranteed to be delivered in causal order. (Likewise with the roles of l and o reversed.) When $l_{i,a}.Dests = \emptyset$, it can be inferred that $M_{i,a}$ is delivered or is transitively guaranteed to be delivered in causal order, to all its destinations. Observe that entries of such format will accumulate. Such entries can be discarded and implicitly inferred as follows.
- **Implicit Tracking 2**: If $a_1 < a_2$ and $l_{i,a_2} \in LOG_j$, then l_{i,a_1} is implicitly or explicitly in LOG_j. Entries of the form $l_{i,a_1}.Dests = \emptyset$ can be inferred by their absence and should not be stored.

Implicit Tracking 1 and Implicit Tracking 2 in a combined form implement Pruning Condition 1 and Pruning Condition 2. When doing Implicit Tracking 1 and Implicit Tracking 2, if explicitly stored information is encountered, it must be deleted as soon as it comes into the causal future of the implicitly tracked information.

4.2. The algorithm

We give the algorithm in Algorithm 2. The following data structures are maintained at each site:

1. $clock_i$: local counter at site s_i for write operations performed by application process ap_i.
2. $Apply_i[1 \ldots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i.
3. $LOG_i = \{\langle j, clock_j, Dests\rangle\}$: the local log (initially set to empty). Each entry indicates a write operation initiated by site s_j at time $clock_j$ in the causal past. *Dests* is the destination list for that write operation. Only necessary destination information is stored.
4. $LastWriteOn_i\langle$variable id, $LOG\rangle$: a hash map of *LOG*s. $LastWriteOn_i\langle h\rangle$ stores the piggybacked *LOG* from the most recent update applied at site s_i for locally replicated variable x_h.

In a write operation, the meta-data per replica site is tailored in lines (2)–(9) to minimize its space overhead. L_w is the working variable used to modify the local LOG_i to send to each replica site. Notice that lines (4)–(6) and lines (10)–(11) prune the destination information using Propagation Condition 2, while lines (29)–(30) use Propagation Condition 1 to prune the redundant information. Also, in lines (7)–(8) and in the PURGE function (see Algorithm 3), entries with empty destination list are kept as long as and only as long as they are the most recent update from the sender. This implements the optimality techniques of Implicit Tracking 2, described before. A_{OPT} is implemented in lines (24)–(25).

Algorithm 3 gives the procedures used by Algorithm Opt-Track (Algorithm 2). Function PURGE removes old records with $\emptyset$ destination lists, per sender process (Implicit Tracking 2). On a read operation of variable x_h, function MERGE merges the piggybacked log of the corresponding write to x_h with the local log LOG_i. In this function, new dependencies get added to LOG_i and existing dependencies in LOG_i are pruned, based on the information in the piggybacked data L_w. The merging implements the optimality techniques of Implicit Tracking 1, described before.

5. Opt-Track-CRP: Adapting Opt-Track algorithm to fully-replicated systems

5.1. Basic idea

Algorithm Opt-Track can be directly applied to fully replicated DSM systems. In the full replication case, as every write operation will be sent to the same set of sites, namely all of them, there is no need to keep a list of the destination information with each

Algorithm 2: Opt-Track Algorithm (Code at site s_i)

WRITE(x_h, v):
1 $clock_i + +$;
2 **for** *all sites* $s_j (j \neq i)$ *that replicate* x_h **do**
3 $L_w := LOG_i$;
4 **for** *all* $o \in L_w$ **do**
5 **if** $s_j \notin o.Dests$ **then** $o.Dests := o.Dests \setminus x_h.replicas$;
6 **else** $o.Dests := o.Dests \setminus x_h.replicas \cup \{s_j\}$;
7 **for** *all* $o_{z,clock_z} \in L_w$ **do**
8 **if** $o_{z,clock_z}.Dests = \emptyset \wedge (\exists o'_{z,clock'_z} \in L_w | clock_z < clock'_z)$ **then** remove $o_{z,clock_z}$ from L_w;
9 send $m(x_h, v, i, clock_i, x_h.replicas, L_w)$ to site s_j;
10 **for** *all* $l \in LOG_i$ **do**
11 $l.Dests := l.Dests \setminus x_h.replicas$;
12 $LOG_i := LOG_i \cup \{\langle i, clock_i, x_h.replicas \setminus \{s_i\}\rangle\}$;
13 PURGE;
14 **if** x_h *is locally replicated* **then**
15 $x_h := v$;
16 $Apply_i[i] + +$;
17 $LastWriteOn_i\langle h\rangle := LOG_i$;

READ(x_h):
18 **if** x_h *is not locally replicated* **then**
19 RemoteFetch[$f(x_h)$] from any site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
20 MERGE(LOG_i, $LastWriteOn_j\langle h\rangle$);
21 **else** MERGE(LOG_i, $LastWriteOn_i\langle h\rangle$);
22 PURGE;
23 return x_h;

On receiving $m(x_h, v, j, clock_j, x_h.replicas, L_w)$ from site s_j:
24 **for** *all* $o_{z,clock_z} \in L_w$ **do**
25 **if** $s_i \in o_{z,clock_z}.Dests$ **then** wait until $clock_z \leq Apply_i[z]$;
26 $x_h := v$;
27 $Apply_i[j] := clock_j$;
28 $L_w := L_w \cup \{\langle j, clock_j, x_h.replicas\rangle\}$;
29 **for** *all* $o_{z,clock_z} \in L_w$ **do**
30 $o_{z,clock_z}.Dests := o_{z,clock_z}.Dests \setminus \{s_i\}$;
31 $LastWriteOn_i\langle h\rangle := L_w$;

On receiving $f(x_h)$ from site s_j:
32 return x_h and $LastWriteOn_i\langle h\rangle$ to s_j;

Algorithm 3: Procedures used in Algorithm 2, Opt-Track Algorithm (Code at site s_i)

PURGE:
1 **for** *all* $l_{z,t_z} \in LOG_i$ **do**
2 **if** $l_{z,t_z}.Dests = \emptyset \wedge (\exists l'_{z,t'_z} \in LOG_i | t_z < t'_z)$ **then**
3 remove l_{z,t_z} from LOG_i;

MERGE(LOG_i, L_w):
4 **for** *all* $o_{z,t} \in L_w$ *and* $l_{s,t'} \in LOG_i$ *such that* $s = z$ **do**
5 **if** $t < t' \wedge l_{s,t} \notin LOG_i$ **then** mark $o_{z,t}$ for deletion;
6 **if** $t' < t \wedge o_{z,t'} \notin L_w$ **then** mark $l_{s,t'}$ for deletion;
7 delete marked entries;
8 **if** $t = t'$ **then**
9 $l_{s,t'}.Dests := l_{s,t'}.Dests \cap o_{z,t}.Dests$;
10 delete $o_{z,t}$ from L_w;
11 $LOG_i := LOG_i \cup L_w$;

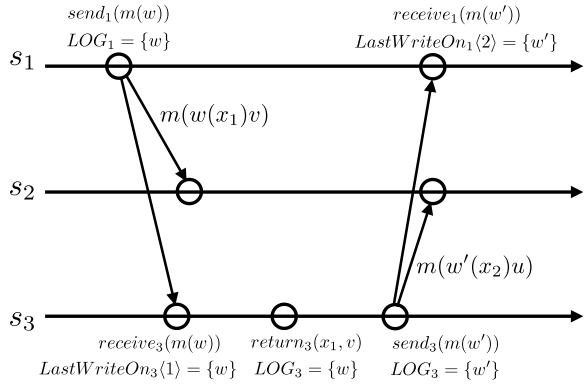


Fig. 2. In fully replicated systems, the local log will be reset after each write operation. Also, when a write operation is applied, only the write operation itself needs to be remembered. For clarity, the *apply* events are omitted in this figure.

write operation. Each time a write operation is sent, all the write operations it piggybacks as its dependencies will share the same set of destinations as the one being sent, and their destination list will be pruned by Propagation Condition 2. Also, when a write operation is received, all the write operations it piggybacks also have the receiver as part of their destinations. So, when checking for the activation predicate at lines (24)-(25) in Algorithm 2, all piggybacked write operations need to be checked. With these additional properties in the full replication scenario, we can represent each individual write operation using only a pair $\langle i, clock_i\rangle$, where i is the site id and $clock_i$ is the local write operation counter at site s_i when the write operation is issued. In this way, we bring the cost of representing a write operation from potentially $O(n)$ down to $O(1)$. This improves the algorithm's scalability when the shared memory is fully replicated.

In fact, Algorithm 2's scalability can be further improved in the fully replicated scenario. In the partially replicated case, keeping entries with empty destination list as long as they represent the most recent applied updates from some site is important, as it ensures the optimality that no redundant destination information is transmitted. However, this will also require each site to almost always maintain a total of n entries. In the fully replicated case, we can also decrease this cost. We observe that, once a site s_3 issues a write operation $w'(x_2)u$, it no longer needs to remember any previous write operations, such as $w(x_1)v$, stored in the local log. This is because all the write operations stored in the local log share the same destination list as w'. Thus, by making sure the most recent write operation is applied in causal order, all the previous write operations sent to all sites are guaranteed to be also applied in causal order. Similarly, after the activation predicate becomes true and the write operation w' is applied at site s_1, only w' itself needs to be remembered in $LastWriteOn_1\langle 2\rangle$. This is illustrated in Fig. 2.

This way of maintaining local logs essentially means that each site s_i now only needs to maintain $d + 1$ entries at any time with each entry incurring only an $O(1)$ cost. Here, d is the number of read operations performed locally since the most recent local write operation (and is equal to the number of write operations stored in the local log). This is because the local log always gets reset after each write operation, and each read operation will add at most one new entry into the local log. Furthermore, if some of these read operations read variables that are updated by the same application process, only the entry associated with the very last read operation needs to be maintained in the local log. Thus, the number of entries to be maintained in the full replication scenario will be at most n.

Furthermore, if the application running on top of the DSM system is write-intensive, then the local log will be reset at the

Table 1
Complexity measures of causal memory algorithms.

Metric	Full-Track	Opt-Track	Opt-Track-CRP	*optP* [3]
Message count	$((p-1)+\frac{n-p}{n})w+2r\frac{(n-p)}{n}$	$((p-1)+\frac{n-p}{n})w+2r\frac{(n-p)}{n}$	$(n-1)w$	$(n-1)w$
Message size	$O(n^2pw+nr(n-p))$	$O(n^2pw+nr(n-p))$ amortized $O(npw+r(n-p))$	$O(nwd)$	$O(n^2w)$
Time complexity	Write $O(n^2)$ Read $O(n^2)$	Write $O(n^2p)$ Read $O(n^2)$	Write $O(d)$ Read $O(d)$	Write $O(n)$ Read $O(n)$
Space complexity	$O(\max(n^2, npq))$	$O(\max(n^2, npq))$ amortized $O(\max(n, pq))$	$O(\max(d, q))$	$O(nq)$

frequency of write operations issued at each site. This means, each site simply cannot perform enough read operations to build up the local log to reach a number of n entries. Even if the application is read-intensive, this is still the case because read-intensive applications usually only have a limited subset of all the sites whose write operations are read. Thus, in practice, the number of entries that need to be maintained in the full replication scenario is much less than n.

5.2. The algorithm

With the above discussion, we give the formal algorithm of a special case of Algorithm 2, optimized for the fully replicated DSMs. The algorithm is listed in Algorithm 4. Each site still maintains the same data structures as in Algorithm 2, the main difference lies in that there is no need to maintain the destination list for each write operation in the local log, and hence the format of the log entries becomes the pair $\langle i, clock_i\rangle$. We also use a boolean, $unionflag_i$, to implement the MERGE function. Algorithm 4 assumes a highly simplified form. However, it is very systematically derived by adapting Algorithm 2 to the fully replicated case. Algorithm 4 is significantly better than the algorithm in [3] in multiple respects.

Function MERGE works similarly to that in Opt-Track. There are two merge cases in Opt-Track-CRP. First, when any log entry $l_{s,t}$ with clock time t for site $s = j$ in LOG_i is older than the piggybacked log entry $\langle j, clock_j\rangle$ (i.e., $t < clock_j$), it implies that this entry information is out of date compared with $\langle j, clock_j\rangle$. It also means that there is no entry in LOG_i whose time clock is greater than $clock_j$. If so, this entry $l_{s,t}$ will be removed from LOG_i and entry $\langle j, clock_j\rangle$ needs to be united with LOG_i. Note that, in full replication, it is impossible that there are multiple entries with the same site id and different clock time marks (i.e., $LOG_i = \{\langle j, clock_j\rangle, \langle j, clock'_j\rangle \ldots\}$). Second, if there is no log entry $l_{s,t}$ in which site id s is equal to j, then, the piggybacked log entry $\langle j, clock_j\rangle$ will also be merged with LOG_i.

Algorithm 4: Opt-Track-CRP Algorithm (Code at site s_i)

WRITE(x_h, v):
1 $clock_i + +$;
2 send $m(x_h, v, i, clock_i, LOG_i)$ to all sites other than s_i;
3 $LOG_i := \{\langle i, clock_i\rangle\}$;
4 $x_h := v$;
5 $Apply_i[i] := clock_i$;
6 $LastWriteOn_i\langle h\rangle := \langle i, clock_i\rangle$;

READ(x_h):
7 MERGE(LOG_i, $LastWriteOn_i\langle h\rangle$);
8 return x_h;

On receiving $m(x_h, v, j, clock_j, L_w)$ from site s_j:
9 **for** *all* $o_{z,clock_z} \in L_w$ **do**
10 wait until $clock_z \le Apply_i[z]$;
11 $x_h := v$;
12 $Apply_i[j] := clock_j$;
13 $LastWriteOn_i\langle h\rangle := \langle j, clock_j\rangle$;

MERGE(LOG_i, $\langle j, clock_j\rangle$):
14 $unionflag_i := 1$;
15 **for** *all* $l_{s,t} \in LOG_i$ *such that* $s = j$ **do**
16 **if** $t < clock_j$ **then**
17 delete $l_{s,t}$ from LOG_i;
18 **else**
19 $unionflag_i := 0$;
20 **if** $unionflag_i$ **then** $LOG_i := LOG_i \cup \{\langle j, clock_j\rangle\}$;

6. Complexity

We analyze the complexity of the algorithms proposed in this paper. Four metrics are used.

- message count: count of the total number of messages generated.
- message size: the total size of all the messages generated by the algorithm. It can be formalized as $\sum_i$*(# type i messages * size of type i messages)*.
- time complexity: the time complexity at each site s_i for performing the write and read operations.
- space complexity: the space complexity at each site s_i for storing local logs and the *LastWriteOn* log.

The following parameters are used in the analysis:

- n: the number of sites in the system
- q: the number of variables in the DSM system
- p: the replication factor, i.e., the number of sites at which each variable is replicated
- w: the number of write operations performed in the DSM system
- r: the number of read operations performed in the DSM system
- d: the number of write operations stored in local log (used only in the analysis of Opt-Track-CRP algorithm). Note that $d \le n$.

Table 1 summarizes the results.

6.1. Full-Track algorithm

Message count complexity. In Full-Track, each write operation will incur $(p-1)+\frac{n-p}{n}$ number of messages. However, since now read operations might need to read from remote site, assuming the variables are evenly replicated across the entire system and the read operations read variables in a truly random manner, then an additional $\frac{2(n-p)}{n}$ number of messages will be generated by a read operation. In total, the message count complexity of Full-Track is $((p-1)+\frac{n-p}{n})w+\frac{2r(n-p)}{n}$.

Message space complexity. Full-Track maintains a local *Write* clock n^2 matrix. Since this clock is piggybacked with each message

containing a write operation or a remote read operation, each message generated by a send event and the remote return event in Full-Track has a size of $O(n^2)$, (whereas each message generated by a fetch event has a small and constant byte count). Thus, the total message size complexity of Full-Track is $npw(n-1) + nr(n-p)$, which is $O(n^2pw + nr(n-p))$.

Time complexity. Each write operation updates the local *Write* clock for each of the p replicas before invoking the `Multicast` primitive. This incurs an $O(p)$ time complexity. There is an added cost of $O(n^2)$ for copying with the probability of $\frac{p}{n}$ the $Write_i$ matrix into the $LastWriteOn_i$ log. For the read operation, merging the *Write* clock associated with the variable to be read with the local clock incurs an $O(n^2)$ time complexity.

Space complexity. The total space cost comes from the local *Write* clock and the *LastWriteOn* log. The local *Write* clock takes $O(n^2)$ space. Each entry in the *LastWriteOn* log that is associated with a locally replicated variable is of size $O(n^2)$. Assuming the variables are evenly distributed across the entire system, each site will replicate a total of $\frac{pq}{n}$ variables. Thus, the *LastWriteOn* log incurs an $O(npq)$ space complexity. The Full-Track algorithm's space complexity is $O(\max(n^2, npq))$.

6.2. Opt-Track algorithm

Message count complexity. As Opt-Track produces the same message pattern as Full-Track, its message count complexity is also the same, being $((p-1) + \frac{n-p}{n})w + \frac{2r(n-p)}{n}$.

Message space complexity. Different from Full-Track, Opt-Track keeps a log of only the necessary write operations from all that happened in the causal past and guarantees that only the necessary destination information are kept, and piggybacked. Each record of a write operation also maintains its destination list, which contains up to p sites. Similar to Full-Track, each message generated by a send event and a remote return event contributes to the main meta-data overhead. The messages generated by a fetch event are small and of constant byte size.

In the KS algorithm, the upper bound on the size of the log and the message overhead is $O(n^2)$ [24]. This has also been shown using an adversarial argument [28], viz., if you cannot decide the distribution of replicas, then partial replication incurs the same costs as does full replication as each process has to manage information on all the replicas. However, Chandra et al. [29,30] showed through extensive simulations that the amortized log size and message overhead size of the KS algorithm is approximately $O(n)$. This is because the optimality conditions implemented ensure that only necessary destination information is kept in the log and purged as soon as possible. This also applies to the Opt-Track algorithm because the same optimization techniques are used. Since the log is piggybacked with each message containing a write operation and each message that returns a variable's value to a remote site, the total message size complexity of the Opt-Track algorithm is $O(n^2pw + nr(n-p))$. However, this is only the asymptotic upper bound. The amortized message size complexity of the Opt-Track algorithm is approximately $O(npw + r(n-p))$.

Time complexity. For a write operation, for each replica the algorithm prunes the destination information stored in the local log accordingly before piggybacking it with the message. This will incur an $O(n^2p)$ time complexity. For the read operation, the `MERGE` operation will merge the log associated with the variable to be read with the local log. If the local logs are maintained in such a way that all the entries $l_{j,t_j} \in LOG_i$ are stored in the ascending order of j and t_j, then the `MERGE` operation can be completed with only one pass through both logs. Thus, the incurred time complexity of a read operation is $O(n^2)$.

Space complexity. The total space cost comes from the local LOG_i log (of size $O(n^2)$) and the *LastWriteOn* log. Since for each of the $\frac{qp}{n}$ locally replicated variables, Opt-Track needs to store the log piggybacked with the most recent write operation updating that particular variable, the size of the *LastWriteOn* log is $O(npq)$. Thus, the space complexity is $O(\max(n^2, npq))$. Still, this is only the asymptotic upper bound. The amortized space complexity will be approximately $O(\max(n, pq))$, since the amortized size of the local log is approximately $O(n)$ on average, instead of $O(n^2)$ [29,30].

6.3. Opt-Track-CRP algorithm

Message count complexity. Each write operation incurs $n-1$ messages, however the read operation will always read from the local copy. The message count complexity of Opt-Track-CRP is $(n-1)w$.

Message space complexity. Being a special case of Opt-Track, Opt-Track-CRP does not keep the destination list for each record of a write operation, nor does it always maintain n entries in the local log at each site (as discussed in Section 5). This means the size of a write operation's record is only $O(1)$ and the size of the local log is only determined by the number of entries in the local log, which is denoted as d in this section. As discussed in Section 5, in practice d is only a small constant number. Thus the size of the log piggybacked with each message containing a write operation is $O(d)$ and the total message size complexity of the Opt-Track-CRP algorithm is $O(nwd)$.

Time complexity. For the write operation, the algorithm rewrites the local log and thus incurs only an $O(1)$ time complexity. The processing on receiving a write broadcast checks for each piggybacked log entry, which is the size of the sender's log, and takes $O(d)$ time. For the read operation, the `MERGE` operation merges the log associated with the variable to be read with the local log. As each log record stored in the *LastWriteOn* hash map always contains only one write operation, the `MERGE` operation can be completed within $O(d)$ time.

Space complexity. The total space cost comes from the *LastWriteOn* log and the local log. The *LastWriteOn* log contains q logs, each containing a single write operation; thus its size is $O(q)$. For the local log, it contains d entries of write operations, thus having a size of $O(d)$. The space complexity of the Opt-Track-CRP algorithm is thus $O(max(d, q))$.

6.4. Analysis

Compared with the existing causal memory algorithms, our suite of algorithms has advantages in several aspects. Similar to the complete replication and propagation causal memory algorithm, *optP*, proposed by Baldoni et al., our algorithm also adopts the optimal activation predicate. However, compared with Opt-Track-CRP, the *optP* algorithm incurs a higher cost in the message size complexity, the time complexity for read and write operations, and the space complexity. This is because the *optP* algorithm requires each site to maintain a *Write* clock of size n.

Compared with other causal consistency algorithms [5–7,12–19], our algorithms have the additional ability to implement causal consistency in partially replicated DSM systems. The performance of Full-Track provides a baseline for measuring the optimality performance of Opt-Track. The performance of algorithm *optP* provides a baseline for measuring the optimality performance of Opt-Track-CRP, which follows the same system model; for the fully replicated case, we do not compare the complexity of our algorithm Opt-Track-CRP with those of these other algorithms because they use a different system model and because the emphasis of this paper is on partial replication protocols. The benefit of doing partial replication compared with full replication lies in multiple aspects, as described in Section 1.

Message count. Of the four metrics in Table 1, message count is the most important. As the formulas indicate, partial replication gives a lower message count than full replication if

$$\left((p-1)+\frac{(n-p)}{n}\right)w+2r\frac{(n-p)}{n}<(n-1)w$$
$$\Rightarrow w>2\frac{r}{n-1}. \quad (1)$$

This is equivalently stated as: partial replication has a lower message count if the write rate (defined as $w_{rate}=\frac{w}{w+r}$) is such that

$$w_{rate}>\frac{2}{1+n}. \quad (2)$$

Message size. Partial replication can also help to reduce the total size of messages transmitted within the system. Although the two partial replication causal memory algorithms proposed in this paper (Full-Track and Opt-Track) might have a higher message size complexity compared with the algorithms for full replication (*optP* and Opt-Track-CRP), this complexity measurement does not take into account the size of the real payload data. In modern social networks, multimedia files like images and videos are frequently shared. These files are much larger than the meta-data control information piggybacked with them. Doing full replication might somewhat improve the file access latency, however it also incurs a large overhead on the underlying system for transmitting and storing these files across different sites.

Let f be the size of an image being written and let b be the number of bytes in an integer.

Under full replication, the net message payload size for the write multicast is $(n-1)f$, and $n(n-1)b$ for the message meta-data overheads in the worst case of Opt-Track-CRP protocol [26]. The read cost is zero. In practice, the size of a write operation's record is only $O(1)$ and the size of the local log is determined only by the number of entries in the local log, denoted as d; d is only a small constant number. Thus, the message size cost introduced by one write operation in full replication is $(n-1)f+db(n-1)$.

Under partial replication using Opt-Track, the net message payload size is $((p-1)+\frac{(n-p)}{n})f$ for the write multicast. The read cost is $(\frac{r}{w})\frac{(n-p)}{n}f$ because there are $\frac{r}{w}$ reads per write, and $\frac{n-p}{n}$ of the reads fetch the file from a remote location. The corresponding message meta-data overheads are $((p-1)+\frac{(n-p)}{n})n^2b+(\frac{r}{w})\frac{(n-p)}{n}(n^2+1)b$ in the worst case. However, it was shown in [27] through extensive simulations that the amortized log size and message overhead size is approximately $O(n)$, not $O(n^2)$. The amortized message meta-data overheads are $((p-1)+\frac{(n-p)}{n})nb+(\frac{r}{w})\frac{(n-p)}{n}(n+1)b$. Thus, partial replication has lower message size in the worst case if

$$\begin{aligned}&(n-1)f+nb(n-1)>\\&\left((p-1)+\frac{(n-p)}{n}\right)f+\left(\frac{r}{w}\right)\frac{(n-p)}{n}f+\\&\left((p-1)+\frac{(n-p)}{n}\right)n^2b+\left(\frac{r}{w}\right)\frac{(n-p)}{n}(n^2+1)b\end{aligned} \quad (3)$$

or, in practice, if

$$\begin{aligned}&(n-1)f+db(n-1)>\\&\left((p-1)+\frac{(n-p)}{n}\right)f+\left(\frac{r}{w}\right)\frac{(n-p)}{n}f+\\&\left((p-1)+\frac{(n-p)}{n}\right)nb+\left(\frac{r}{w}\right)\frac{(n-p)}{n}(n+1)b.\end{aligned} \quad (4)$$

7. Simulation system model

We consider an asynchronous distributed system composed of a finite set of asynchronous processes running on multiple sites interconnected through a communication network over a wide area.

Table 2
Message meta-data structures in partial replication protocols.

	Full-Track	Opt-Track
SM (Multicast)	x_h, v, *Write*	x_h, v, $Site_{id}$, *clock*, L_w
FM (Fetch)	x_h, v	x_h, v
RM (Remote return)	v, *LastWriteOn*$\langle h\rangle$	v, *LastWriteOn*$\langle h\rangle$

To simplify and without loss of generality, we assume that there is only one process on each site. Each site has a local memory and can communicate by asynchronous message passing through TCP channels of the underlying network. The communication network is reliable and transmits messages in FIFO order without omissions or duplications.

7.1. Process Model

7.1.1. Partial replication

A process consists of two major subsystems viz., the *application subsystem* and *message receipt subsystem*. The application subsystem is responsible for the functionality of scheduling operation events (write/read) including two major functions, viz., *Write* and *Read*. The message response subsystem is responsible for responding to remote request service, including two major functions, viz., *Applying Multicast* and *Responding Fetch*. The application subsystem executes write and read events. It also maintains a floating point local clock to generate operation patterns based on a temporal schedule. For a write operation $w(x_h)v$, the application subsystem delivers the message $m(w(x_h)v)$ and the corresponding meta-data – local log L_w (in **Opt-Track protocol**) or local *Write* clock (in **Full-Track protocol**) – to other replicas. For a read operation $r(x_h)$, the application subsystem returns the local variable x_h's value or sends a fetch message $fetch(x_h)$ to a predesignated site to get the remote variable x_h's value as well as the corresponding meta-data *LastWriteOn*$\langle h\rangle$.

The message receipt subsystem monitors two kinds of incoming messages. First, for a multicast message $m(w(x_h)v)$, the message receipt subsystem determines when to apply a new value v to the variable x_h in a causally consistent manner and then update the meta-data *LastWriteOn*$\langle h\rangle$. Second, for a remote fetch message $m(fetch(x_h))$, it simply replies with the local value of the variable x_h and the corresponding meta-data *LastWriteOn*$\langle h\rangle$ to the requesting site.

Message structure. A message is the fundamental entity that delivers information from a sender process to one or more receiver processes. The structures of different kinds of messages typically follow the data structures shown in Table 2 for the algorithms. In partial replication protocols, there are three distinct messages. SM corresponds to a multicast message generated by *send event* to deliver the information of updating variable's value. FM is a fetch message caused by a *fetch event*. RM represents a remote return message to respond to a remote read operation.

The Full-Track protocol imposes an $n\times n$ *Write* matrix structure as part of the piggybacked meta-data in SM and RM messages. The Opt-Track protocol utilizes a list log to maintain causal ordering information in SM and RM messages.

7.1.2. Full replication

Similarly, a process also consists of two major subsystems viz., the *application subsystem* and *message receipt subsystem*, in the full replication protocols. The functionalities of the two subsystems are very similar to those of subsystems in partial replication protocols. The application process consists of *Write* and *Read* functions. Specially, the function $Read(x_h)$ only needs to merge the local piggybacked log *LastWriteOn*$\langle h\rangle$ into the local log *LOG* and then return the local value of x_h. Besides, the message receipt subsystem

only handles the function of *Applying Multicast* to determine when to apply a write update.

Message structure. There is only one message type – SM message corresponding to a write operation for multicasting – in Opt-Track-CRP. SM message is represented by $m(x_h, v, Site_{id}, clock, LOG)$.

7.2. Simulation parameters

The system parameters whose effects we examine on the performance of the Opt-Track partial replication protocol and the Opt-Track-CRP full replication protocol are n, q, p, w, r, and w_{rate}. We make some notes as follows.

- **Number of Processes** (n): Due to our hardware limitation, we can simulate up to 40 processes in JDK8.
- **Number of Variables** (q): It is the number of variables in the distributed shared memory system. In a super geo-replicated cloud storage, q is unbounded. Due to the memory limitation, q we used in the benchmark experiment is one hundred.
- **Replication Factor** (p): It is defined as the number of sites at which each variable is replicated. In full replication protocols, $p = n$. In partial replication protocols, we set p equal to $0.3 \times n$.

7.3. Process execution

All the processes in the system are symmetric and generate operation events (write event or read event) according to an event schedule planned in advance. The event schedule is randomly generated. The time interval between two events is given from 5 to 2005 ms. The processes in the distributed system execute concurrently. However, simulating each process as an independent process at a site invoked interprocess communication.

When a process gets initialized, it first invokes the message receipt subsystem. Then, the system executes *ScheduledExecutorService* in JDK to drive the application subsystem which extends *TimerTask* class – a JDK scheduling service to dispose of the scheduling operation events. In the simulation, the system relies on TCP channels to deliver messages. While TCP exploits *slow start* to retransmit the lost packets, which can lead to high value of transmission time, it guarantees that messages can be successfully transmitted without any loss.

An application subsystem stops generating operation events once it runs out of all the scheduling events and flags its status as finished. The simulation stops when all the application subsystems have their status set to 'finished'.

8. Simulation results

The implementations of the four causal consistency replicated protocols – Full-Track, Opt-Track, Opt-Track-CRP, and *optP* [3] – were realized in the framework presented in Section 7. The performance metrics we used are to measure the ratio of the total message cost for Full-Track vs. Opt-Track and Opt-Track-CRP vs. *optP*, and the average size of the messages transmitted in different causal consistency replicated protocols.

We report two experiments for each protocol, in each of which we vary one of the two parameters n and w_{rate}, respectively. For each combination of parameters in each experiment, multiple runs were performed for each protocol. The experimental results of all the runs did not have more than one percent variation. Thus, only the mean of the multiple runs is represented for each combination. Each simulation execution runs $600n$ operation events totally. Experimental data was stored after the first 15% operation events to eliminate the side effects of startup.

Table 3
Average SM and RM space overhead for Opt-Track and Full-Track (KB).

		w_{rate}	The number of processes, n				
			5	10	20	30	40
Opt-Track	SM	0.2	0.489	0.828	1.512	2.241	2.783
		0.5	0.464	0.715	1.125	1.442	1.976
		0.8	0.450	0.627	0.914	1.194	1.475
	RM	0.2	0.432	0.774	1.530	2.351	3.184
		0.5	0.436	0.702	1.235	1.656	2.197
		0.8	0.555	0.632	0.948	1.288	1.599
Full-Track	SM	0.2	0.518	1.252	3.870	8.028	13.547
		0.5	0.522	1.271	3.975	8.127	14.033
		0.8	0.524	1.275	3.988	8.410	14.157
	RM	0.2	0.493	1.220	3.817	7.959	13.461
		0.5	0.497	1.205	3.941	8.117	13.983
		0.8	0.499	1.250	3.966	8.369	14.099

8.1. Partial replication protocols: Meta-data size

As shown in Table 2, in Full-Track and Opt-Track, SM and RM messages contribute to the meta-data overheads, whereas FM messages are of small and constant byte size.

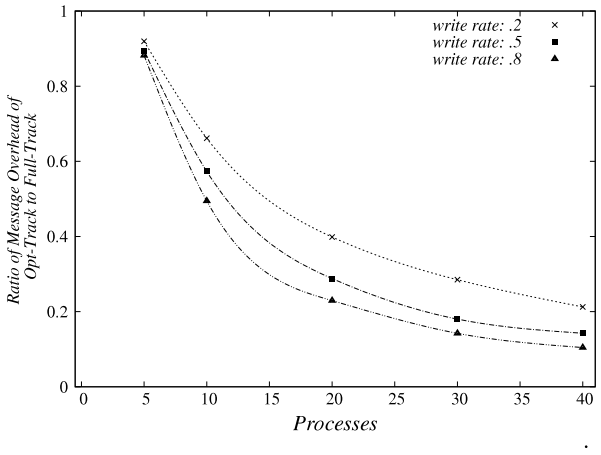
8.1.1. Scalability as a function of n

The number of processes was varied from 5 up to 40. The w_{rate} is set to be 0.2 (lower write rate), 0.5 (medium write rate), and 0.8 (higher write rate), respectively. The results for the ratio of message space overhead (meta-data size) of Opt-Track to Full-Track are shown in Fig. 3(a). With increasing n, the space overhead ratio rapidly decreases. For the case of 40 processes, for all the simulations of Opt-Track, the overheads are only around $10 \sim 20\%$ those of Full-Track on different write rates. For the case of 5 processes, the overheads reported for Opt-Track for different write rates are around 90% of the ones of Full-Track, but the overhead of Full-Track itself is low for such a parameter setting. It can also be seen from Fig. 3(a) that a higher write rate magnifies the difference of the message space overhead between Opt-Track and Full-Track.

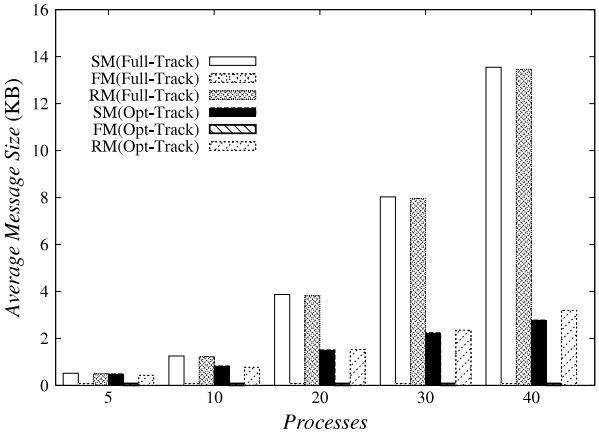
8.1.2. Impact of write rate w_{rate}

The results for the average message space overhead are shown in Figs. 3(b), 3(c), and 3(d) for different write rates, respectively. As discussed before, the average message overheads of FM in Opt-Track and Full-Track protocols are constant, very small, and the same, regardless of write rates. In Full-Track protocol, the average message space overheads of SM and RM quadratically increase with n based on our previous discussion. However, the increasingly lower overheads of SM and RM in Opt-Track protocol can be seen from the results. Their overheads appear almost linear in n. This observation can be explained as follows: Although more explicit information of type "s_i is a destination of message m" needs to be maintained in the logs, each log also involves more implicit information. Additional implicit information provides incentive for the Propagation Constraints to merge and prune the logs when SM or RM messages are received. The observation from Figs. 3(b) to 3(d) demonstrates the scalability of Opt-Track under partial replication.

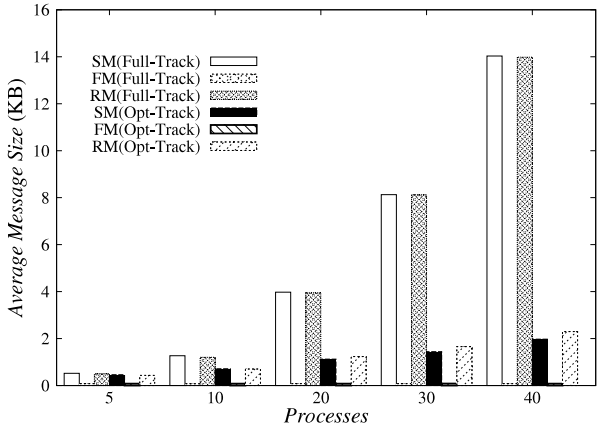
Furthermore, under the same number of processes, we also compare the average SM and RM message sizes in different write rates. (The FM message size is an invariant constant count that is independent of n and w_{rate}. In Full-Track and Opt-Track, their FM sizes are the same since they use the same data structure for FM message.) The analytic data is listed in Table 3 according to Figs. 3(b) to 3(d). The average SM and RM overheads decrease as the write rate increases in Opt-Track Protocol. The reason can be explained as follows. A read operation will invoke a *MERGE*



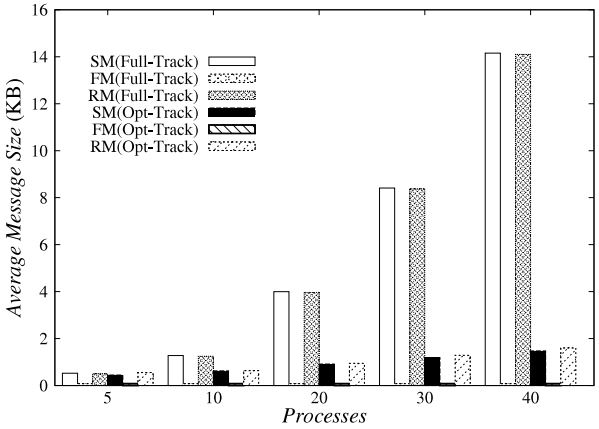
(a) Total message meta-data space overhead as a function of n and w_{rate} in partial replication protocols.



(b) Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in partial replication protocols.



(c) Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in partial replication protocols.



(d) Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in partial replication protocols.

Fig. 3. Meta-data space overhead in partial replication protocols.

function to merge the piggybacked log of the corresponding write to that variable with the local log *LOG*. Thus, a higher read rate may increase the likelihood that the size of *LOG* is enlarged. Furthermore, although a write operation results in the increase of explicit information, it comes with a *PURGE* function to prune the redundant information, so that the size of *LOG* could be decreased. Therefore, a higher write rate with a corresponding lower read rate results in fewer *MERGE* and more *PURGE* operations generated. The simulation results show that Opt-Track has a better utilization of network capacity in write-intensive workloads than in read-intensive ones. On the other hand, in Full-Track, although the average SM and RM overheads increase as the write rate does, the increase percentage is only 3% $\sim$ 1%.

From the above analysis, it can be concluded that the implementation of the Opt-Track protocol has a better network capacity utilization and better scalability than Full-Track. These improvements increase in a higher write-intensive workload.

8.2. Full replication protocols: Meta-data size

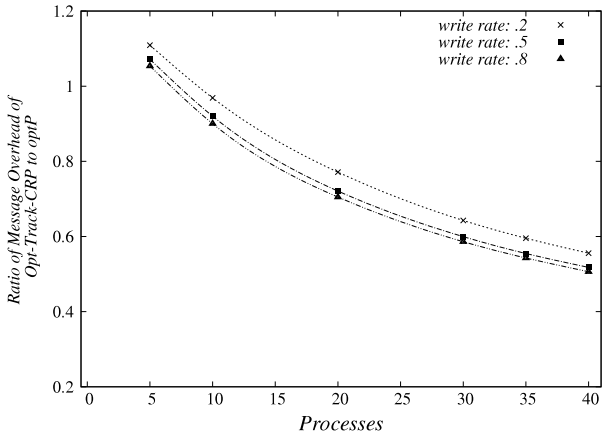
The Opt-Track-CRP protocol and the *optP* protocol both use only SM messages, and no FM or RM messages.

8.2.1. Scalability as a function of n

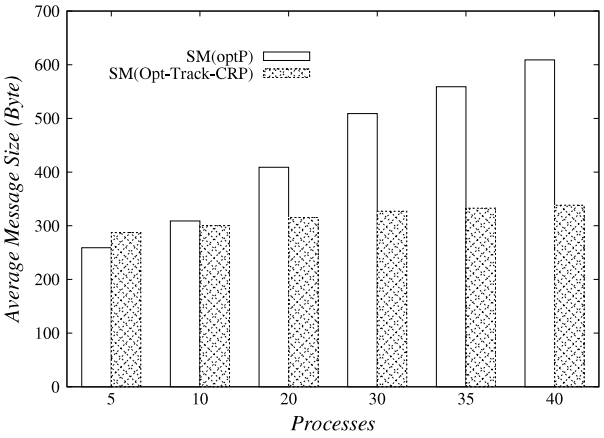
The results for the ratio of message space overhead of Opt-Track-CRP to *optP* are shown in Fig. 4(a). With increasing n, the ratio of total SM space overhead of Opt-Track-CRP vs. *optP* decreases. For the case of 5 processes, the total SM overheads for Opt-Track-CRP are consistently higher by around 10% of those for *optP* on a variety of write rates. For the case of 10 processes, the SM space overhead for Opt-Track-CRP is still close to that for *optP* in a lower write rate 0.2. But their space overhead ratio is down to 90% in a higher write rate 0.8. When the number of processes is up to 40, the SM space overheads for Opt-Track-CRP are around 50% to 55% for different write rates.

8.2.2. Impact of write rate w_{rate}

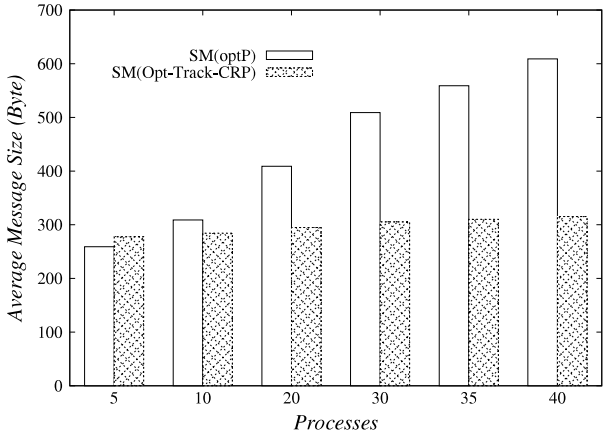
As with partial replication protocols, a higher write rate makes the total message space overhead ratio of Opt-Track-CRP vs. *optP* smaller. The results for the average SM space overhead are shown in Figs. 4(b), 4(c), and 4(d) in terms of different write rates. As mentioned before, the average SM space complexity of Opt-Track-CRP is $O(d)$ but that of *optP* is $O(n)$. According to Figs. 4(b) to 4(d), Table 4 presents the analytic data. Obviously, the average SM space overhead of *optP* only depends on the number of processes n, irrespective of w_{rate}. However, under the same number of processes, the SM space overheads of Opt-Track-CRP decrease slightly with increasing w_{rate}. This can be explained as follows: In Opt-Track-CRP protocol, a write operation does not make the local log size larger than one and does not change the remote log size at a receiving site. But a read operation might incur a growth in the local log size when it often reads different variables updated via other remote sites. Therefore, lower write rate (corresponding



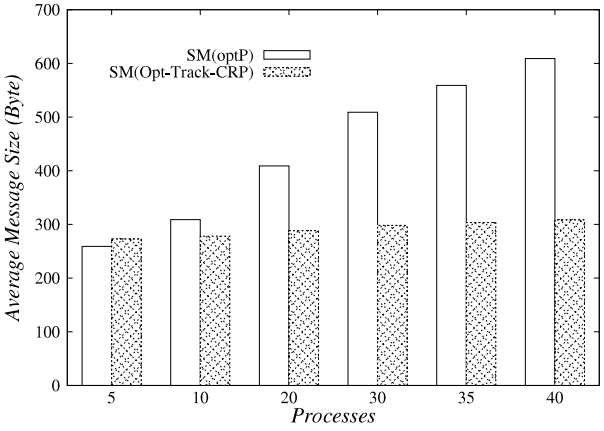
(a) Total message meta-data space overhead as a function of n and w_{rate} in full replication protocols.



(b) Average message meta-data space overhead as a function of n with lower w_{rate} (0.2) in full replication protocols.



(c) Average message meta-data space overhead as a function of n with medium w_{rate} (0.5) in full replication protocols.



(d) Average message meta-data space overhead as a function of n with higher w_{rate} (0.8) in full replication protocols.

Fig. 4. Meta-data space overhead in full replication protocols.

Table 4
Average SM space overhead for Opt-Track-CRP (byte).

n	$w_{rate} = .2$	$w_{rate} = .5$	$w_{rate} = .8$	*optP*
5	287.3	277.5	272.9	259
10	300.3	284.3	278.2	309
20	315.5	294.9	288.3	409
30	327.1	305.2	298.4	509
35	332.8	310.1	303.4	559
40	338.4	315.3	308.4	609

Table 5
Total message count for Full Replication (Opt-Track-CRP) VS. Partial Replication (Opt-Track).

n	Full replication			Partial replication		
	(0.2)	(0.5)	(0.8)	(0.2)	(0.5)	(0.8)
5	2036	4960	8004	3208	3463	3764
10	8910	22,266	35,892	8297	10,234	12,156
20	38,057	95,114	151,905	22,808	35,668	48,128
30	86,826	217,181	347,304	42,600	75,679	108,810
40	156,156	390,039	624,390	69,405	130,572	192,883

to higher read rate) would cause higher meta-data overhead than higher write rate. In other words, Opt-Track-CRP protocol has a better utilization of network capacity in write-intensive workloads than in read-intensive ones.

From the experimental analysis in full replication, we can conclude that Opt-Track-CRP protocol has a better scalability and utilization than *optP*, especially in write-intensive workloads.

8.3. Partial replication vs. full replication: Message count

Compared with the existing causal distributed shared memory protocols, our suite of protocols [26] has the additional ability to implement causal consistency in partially replicated distributed shared memory systems. Table 5 shows the results of running the same operation event scheduling using Opt-Track-CRP and Opt-Track, respectively. It presents the total message counts with different write rates in full replication and partial replication. Except for when n = 5 and $w_{rate} = 0.2$, the message counts for partial replication are always less than the ones for full replication. The results in Table 5 are in line with the necessary condition – Eq. (2).

8.4. Partial replication vs. full replication: Message size

Consider a system with n = 40, $p = 12$ and $w_{rate} = 0.5$. A extra_large/large/medium/small file of 100KB/10KB/1KB/0.1KB, respectively, has to be stored and transmitted in a write operation. Assume that one word holds 4 bytes (b=4B).

Table 6 summarizes the total message space overheads in the worst cases, based on Eq. (3). Although full replication has lower meta-data overheads, partial replication has smaller total message space size when f is large (10KB or 100KB).

Table 7 shows the total message space overheads in the practical amortized sense, based on Eq. (4). Note that $d < n$. As the

Table 6
Total message size for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n = 40$, $p = 12$, $w_{rate} = 0.5$, in the worst case.

	f = 100 KB	f = 10 KB	f = 1 KB	f = 0.1 KB
Full replication	3900 KB + 6.09 KB	390 KB + 6.09 KB	39 KB + 6.09 KB	3.9 KB + 6.09 KB
Partial replication	1240 KB + 77.5 KB	124 KB + 77.5 KB	12.4 KB + 77.5 KB	1.24 KB + 77.5 KB

Table 7
Total message size for full replication (Opt-Track-CRP) and partial replication (Opt-Track), when $n = 40$, $p = 12$, $w_{rate} = 0.5$, in practice.

	f = 100 KB	f = 10 KB	f = 1 KB	f = 0.1 KB
Full replication	3900 KB+0.152d KB	390 KB+0.152d KB	39 KB+0.152d KB	3.9 KB+0.152d KB
Partial replication	1240 KB + 1.94 KB	124 KB + 1.94 KB	12.4 KB + 1.94 KB	1.24 KB + 1.94 KB

experiment results show in Table 4, the average meta-data overhead is 315 B for Opt-Track-CRP; therefore $d = 2$. Whereas the average meta-data overhead for Opt-Track is around 2.09 KB. Thus, the experimental results closely corroborate the above theoretical analysis. Apparently, partial replication, in practice, has lower total message space overhead than full replication, no matter what size of f. This makes partial replication a viable alternative to full replication in DSM systems. Although the meta-data size of full replication is still less than that of partial replication, the difference is far less than the message payload size variation between full replication and partial replication.

9. Discussion

We compared the performance of Full-Track and Opt-Track (analytically and via simulations). These are the first algorithms for causal consistency using partial replication. We also compared analytically and via simulations the performance of the full replication protocol Opt-Track-CRP with that by Baldoni et al. [3] which uses the same system model [2]. We do not compare with full replication geo-replicated protocols such as COPS [12], Eiger [19], Orbe [16] and GentleRain [17] (which are all full replication protocols) because the emphasis of our work is on partial replication protocols and because we use a different system model.

This paper targeted large-scale DSM systems. Our system model is somewhat different from that of cloud and geo-replicated stores, which use a two-level architecture wherein each data center partitions the data among servers and supports thousands of application processes. Both clients and data storages track causal dependencies. As future work, we would like to adapt our partial replication protocols to the cloud and geo-replicated store architectures, where current state-of-the-art uses only full replication protocols.

Algorithms Opt-Track and Opt-Track-CRP are optimal in the size of meta-data stored locally and piggybacked on messages, under our system model, wherein a single sequential application process is assumed to run at a single site (partial replica). This limits the amount of information that needs to be tracked. We note that this does not scale well to the architecture of partitioned and geo-replicated data stores which support multiple application processes per site (replica). In our algorithms, each site, which is supposed to be a data center composed of a large number of servers, would need to handle the meta-data used to enforce causality in a centralized manner within that data center. This can be done by using a centralized component (which might be a bottleneck) or by using coordination mechanisms between servers in a data center (which might be expensive). Assuming a centralized component greatly simplifies the design but has a drawback that all operations executed by the multiple application processes at any site become causally related under the program order relation. This artificially introduces false causal dependencies among unrelated operations of independent application processes. Nevertheless, the value of our algorithms is that they are optimal under the system model (same as in [2,3]), and represent the first algorithms assuming partial replication. Making them scale to geo-replicated stores without introducing artificial causal dependencies is a future challenge.

In future work, it is worth investigating how the *writing semantics* heuristic can be leveraged [31]. This heuristic is orthogonal to the optimality techniques proposed here, and allows an out-of-order write to be applied with the assumption that it immediately overwrites the causally earlier write that was never applied. We would also like to extend our optimality techniques to transactional semantics [12,15,19,32].

Some of the algorithms for fully replicated data stores provide causal+ consistency, or *convergent consistency* [12–15,19]: here, once updates cease, all processes will eventually read the same value (or set of values) of each variable. This provides liveness guarantees. We can provide causal+ consistency for our partially replicated system by using the rule of "last writer wins" when there are concurrent updates, as used in [12].

In our algorithms for partially replicated systems, a read may be non-local. This can affect availability if the process read-from is down. If a non-local read does not respond in a timeout period, then a secondary process is contacted. This provides better availability in light of the CAP Theorem.

The limitation of partial replication concerns relatively higher meta-data overheads against full replication. However, when considering total data size, they can be negligible compared to much higher payload overheads. Even when the payload data size is as small as 100 B (the average size of a plain-text tweet is around 50 B $\sim$ 100 B), Table 7 has demonstrated that partial replication has an advantage over full replication in network capacity. Opt-Track has the apparent merit of targeting large-scale DSM systems, accommodating replications of relatively medium-size data files ($>$ 100 KB). Our system model is especially suitable for multimedia object oriented social networks. For a real world example, 'flickr' is a photo-sharing social community, whose average file size is about 0.6 MB. Our model not only provides causal consistency to improve user experience in the correct order, but also utilizes partial replication to lower the data replication network capacity.

10. Conclusions

We considered the problem of providing causal consistency in large-scale DSM systems under the assumption of partial replication. This work provides the first evidence that explores the causal consistency problem for partially replicated systems and fills in a gap in the literature on causal consistency in distributed shared memory systems. We proposed two algorithms to solve the problem. The first algorithm, Full-Track, is optimal in the sense that each update is applied at the earliest instant while removing false causality in the system. The second algorithm, Opt-Track, is additionally optimal in the sense that it minimizes the size of meta-information carried on messages and stored in local logs.

Simulations showed that it is highly successful in trimming the size of meta-data. We discussed the conditions under which partial replication can provide less overhead (transmission and storage) than the full replication case. In addition, as a derivative of the second algorithm, we proposed an optimized algorithm, Opt-Track-CRP, that reduces the message overhead, the processing time, and the local storage cost at each site in the fully replicated scenario. This algorithm is better than the Baldoni et al. algorithm [3] and the Ahamad et al. algorithm [2].

We then conducted a performance analysis of the message space and message count complexity of the algorithms under a wide range of system conditions using simulations. The simulations considered two partial replication protocols (Full-Track and Opt-Track) and two full replication protocols (Opt-Track-CRP and *optP*), and examined the performance by varying the write rate and the number of processes. Opt-Track was seen to show significant gains over Full-Track in partial replication. In full replication, the results also supported that Opt-Track-CRP performed better than *optP* in scalability and network capacity utilization. In particular, as the size of the system increased to 40 processes, the two optimal protocols performed very well and have lower meta-data overheads under high write-intensive workloads. This paper is also the first such work that explored the trade-off between partial replication and full replication analytically. We showed the advantage of partial replication and provided the conditions under which partial replication can provide less overhead (transmission and storage) than full replication.

Our future work will focus on extending Opt-Track into a heterogeneous geo-replicated system. It will implement a hierarchical partial replication protocol for causal consistency. By introducing an optimal replication strategy, we intend to propose a dynamic, data-driven, optimized replication cost mechanism applied to such a system to ensure the desired data access availability.

References

[1] A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, first ed., Cambridge University Press, New York, NY, USA, 2011. http://dx.doi.org/10.1017/CBO9780511805318.

[2] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto, Causal memory: definitions, implementation and programming, Distrib. Comput. 9 (1) (1995) 37–49.

[3] R. Baldoni, A. Milani, S.T. Piergiovanni, Optimal propagation-based protocols implementing causal memories, Distrib. Comput. 18 (6) (2006) 461–474. http://dx.doi.org/10.1007/s00446-005-0128-5.

[4] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[5] P. Mahajan, L. Alvisi, M. Dahlin, Consistency, availability, and convergence, UTCS TR-11-22 Austin, TX, USA, 2011.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, J. Zheng, PRACTI Replication, USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2006.

[7] K. Petersen, M. Spreitzer, D.B. Terry, M. Theimer, A.J. Demers, Flexible update propagation for weakly consistent replication, in: Proceedings of the Sixteenth ACM Symposium on Operating System Principles, St. Malo, France, October 5–8, 1997, pp. 288–301 http://dx.doi.org/10.1145/269005.266711.

[8] R. Ladin, B. Liskov, L. Shrira, S. Ghemawat, Providing high availability using lazy replication, ACM. Trans. Comput. Syst. 10 (4) (1992) 360–391. http://dx.doi.org/10.1145/138873.138877.

[9] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2) (2002) 51–59. http://doi.acm.org/10.1145/564585.564601.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (6) (2007) 205–220. http://doi.acm.org/10.1145/1294261.1294281.

[11] P.A. Bernstein, S. Das, Rethinking eventual consistency, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2013, pp. 923-928.

[12] W. Lloyd, M.J. Freedman, M. Kaminsky, D.G. Andersen, Don't settle for eventual: scalable causal consistency for wide-area storage with cops,in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, in: SOSP '11, ACM, New York, NY, USA, 2011, pp. 401–416. http://doi.acm.org/10.1145/2043556.2043593.

[13] S. Almeida, J.a. Leitão, L. Rodrigues, Chainreaction: a causal+ consistent datastore based on chain replication, in: Proceedings of the 8th ACM European Conference on Computer Systems, in: EuroSys '13, ACM, New York, NY, USA, 2013, pp. 85–98. http://doi.acm.org/10.1145/2465351.2465361.

[14] P. Bailis, A. Fekete, A. Ghodsi, J.M. Hellerstein, I. Stoica, The potential dangers of causal consistency and an explicit solution, in: Proceedings of the Third ACM Symposium on Cloud Computing, in: SoCC '12, ACM, New York, NY, USA, 2012, pp. 22:1–22:7. http://doi.acm.org/10.1145/2391229.2391251.

[15] P. Bailis, A. Ghodsi, J.M. Hellerstein, I. Stoica, Bolt-on causal consistency, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '13, ACM, New York, NY, USA, 2013, pp. 761–772. http://doi.acm.org/10.1145/2463676.2465279.

[16] J. Du, S. Elnikety, A. Roy, W. Zwaenepoel, Orbe: scalable causal consistency using dependency matrices and physical clocks, in: Proceedings of the 4th Annual Symposium on Cloud Computing, in: SOCC '13, ACM, New York, NY, USA, 2013, pp. 11:1–11:14. http://dx.doi.org/10.1145/2523616.2523628.

[17] J. Du, C. Iorgulescu, A. Roy, W. Zwaenepoel, Gentlerain: cheap and scalable causal consistency with physical clocks, in: Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03–05, 2014, 2014, pp. 4:1–4:13. http://dx.doi.org/10.1145/2670979.2670983.

[18] K. Lady, M. Kim, B.D. Noble, Declared causality in wide-area replicated storage, in: 33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6–9, 2014. pp. 2–7. http://dx.doi.org/10.1109/SRDSW.2014.27.

[19] W. Lloyd, M.J. Freedman, M. Kaminsky, D.G. Andersen, Stronger semantics for low-latency geo-replicated storage, in: Proceeding of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX, Lombard, IL, 2013, pp. 313–328.

[20] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, N.M. Preguiça, Swiftcloud: fault-tolerant geo-replication integrated all the way to the client machine, CoRR,2013. http://dx.doi.org/10.1109/SRDSW.2014.33.

[21] T. Crain, M. Shapiro, Designing a causally consistent protocol for geo-distributed partial replication, in: Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data, in: PaPoC '15, ACM, New York, NY, USA, 2015, pp. 6:1–6:4. http://dx.doi.org/10.1145/2745947.2745953.

[22] M. Raynal, A. Schiper, S. Toueg, The causal ordering abstraction and a simple way to implement it, Inform. Process. Lett. 39 (6) (1991) 343–350.

[23] A.D. Kshemkalyani, M. Singhal, An optimal algorithm for generalized causal message ordering, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, in: PODC '96, ACM, New York, NY, USA, 1996, p. 87. http://dx.doi.org/10.1145/248052.248064.

[24] A.D. Kshemkalyani, M. Singhal, Necessary and sufficient conditions on information for causal message ordering and their optimal implementation, Distrib. Comput. 11 (2) (1998) 91–111. http://dx.doi.org/10.1007/s004460050044.

[25] M. Shen, A.D. Kshemkalyani, T. Y. Hsu, OPCAM: Optimal Algorithms Implementing Causal Memories in Shared Memory Systems, in: Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4–7, 2015. pp. 16:1–16:4. http://dx.doi.org/10.1145/2684464.2684483.

[26] M. Shen, A.D. Kshemkalyani, T.Y. Hsu, Causal consistency for geo-replicated cloud storage under partial replication, in: IPDPS Workshops, IEEE, 2015, pp. 509–518. http://dx.doi.org/10.1109/IPDPSW.2015.68.

[27] T.Y. Hsu, A.D. Kshemkalyani, Performance of causal consistency algorithms for partially replicated systems, in: IPDPS Workshops, IEEE, 2016, pp. 525–534. http://dx.doi.org/10.1109/IPDPSW.2016.148.

[28] J. Hélary, A. Milani, About the efficiency of partial replication to implement distributed shared memory, International Conference on Parallel Processing, ICPP 2006, 14–18 August 2006, Columbus, Ohio, USA, pp. 263–270. http://dx.doi.org/10.1109/ICPP.2006.15.

[29] P. Chandra, P. Gambhire, A.D. Kshemkalyani, Performance of the optimal causal multicast algorithm: a statistical analysis, IEEE Trans. Parallel Distrib. Syst. 15 (1) (2004) 40–52. http://dx.doi.org/10.1109/TPDS.2004.1264784.

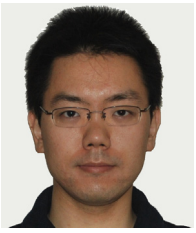
[30] P. Chandra, A.D. Kshemkalyani, Causal multicast in mobile networks, in: 12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems MASCOTS 2004, 4–8 October 2004, Volendam, the Netherlands, pp. 213–220. http://dx.doi.org/10.1109/MASCOT.2004.1348235.

[31] M. Raynal, M. Ahamad, Exploiting write semantics in implementing partially replicated casual objects, in: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 1998, pp. 157–163. http://10.1109/EMPDP.1998.647193.

[32] V. Padhye, G. Rajappan, A. Tripathi, Transaction management using causal snapshot isolation in partially replicated databases, in: Proceedings of the IEEE Symposium on Reliable Distributed Systems, IEEE Computer Society, 2014, pp. 105–114. http://dx.doi.org/10.1109/SRDS.2014.30.

Ta-Yuan Hsu is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Illinois at Chicago, USA. His research interests include distributed algorithms, social networks and causal memory. He is a Student Member of the IEEE.

Ajay D. Kshemkalyani received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the M.S. and Ph.D. degrees in computer and information science from The Ohio State University in 1988 and 1991, respectively. He spent several years at IBM Research Triangle Park working on various aspects of computer networks, before joining academia. He is currently a professor in the Department of Computer Science at the University of Illinois at Chicago. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems. In 1999, he received the US National Science Foundation Career Award. He has served on the editorial board of the Elsevier journal Computer Networks, and the IEEE Transactions on Parallel and Distributed Systems. He has coauthored a book entitled Distributed Computing: Principles, Algorithms, and Systems (Cambridge University Press, 2011). He is a distinguished scientist of the ACM and a senior member of the IEEE.



Min Shen received the B.S. degree in computer science from Nanjing University in 2009, and the Ph.D. degree in computer science from University of Illinois at Chicago in 2014. His research interests include distributed algorithms, predicate detection and wireless sensor networks. He is currently working at LinkedIn on various aspects of Hadoop ecosystem.