Big Data Research 9 (2017) 47–56

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Partial Rollback-based Scheduling on In-memory Transactional Data Grids

Junwhan Kim

ARTICLE INFO

Article history:
Received 2 June 2016
Received in revised form 17 January 2017
Accepted 8 June 2017
Available online 24 August 2017

Keywords:
In-memory data grids
Distributed software transactional memory
Transactional scheduling

ABSTRACT

In-memory transactional data girds, often referred to as NoSQL data grids demand high concurrency for scalability and high performance in data-intensive applications. As an alternative concurrency control model, distributed transactional memory (DTM) promises to alleviate the difficulties of lock-based distributed synchronization. However, if a transaction aborts, DTM suffers from additional communication delays to remotely request and retrieve all its objects again, resulting in degraded performance. To avoid unnecessary aborts, the multi-versioning (MV) model of using multiple object versions in DTM can be considered. MV transactional memory inherently guarantees commits of read-only transactions, but limits concurrency of write transactions. We present a new transactional scheduler, called partial rollback-based transactional scheduler (or PTS), for a multi-versioned DTM model. The model supports multiple object versions to exploit concurrency of read-only transactions, and detects conflicts of write transactions at an object level. Instead of aborting a transaction, PTS assigns backoff times for conflicting transactions, and the transaction is rolled-back partially. We implemented PTS on Infinispan, and conducted comprehensive experimental studies on no and partial replication models. Our implementation reveals that PTS improves transactional throughput over MV-Transactional Forwarding Algorithm without PTS and a scalable one-copy serializable partial replication protocol (SCORe) by as much as 2.4× and 1.3×, respectively.

Published by Elsevier Inc.

1. Introduction

In-memory data grids are increasingly common recently because of their support for dynamic scalability and high performance for data-intensive analytics. Some of the products such as Red Hat's Infinispan, Oracle's Coherence, and Apache Cassandra [23] have been proposed for extreme data management. For transactional processing on in-memory data grids, lock-based concurrency controls have been used but suffered from programmability, scalability, and composability challenges [24]. Transactional memory (TM) promises to alleviate these difficulties.

With TM, code that read/write shared objects is organized as transactions, which optimistically execute, while logging changes made to objects. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes (thanks to the log). Sometimes, a *transactional scheduler* is also used, which determines an ordering of concurrent transactions so that conflicts are either avoided altogether or minimized, thereby reducing aborts and improving performance. In addition to a simple programming model, TM provides performance comparable to lock-based synchronization [37] and is composable. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination. See [24] for an excellent tutorial on TM. Distributed STM (or DTM) has been similarly motivated as an alternative to distributed lock-based concurrency control [8,35,36].

With a single copy for each object, i.e., *single-version* STM (SV-STM), when a read/write conflict occurs between two transactions, the contention manager resolves the conflict by aborting one and allowing the other to commit, thereby maintaining the consistency of the (single) object version. SV-STM is simple, but suffers from large number of aborts [31]. In contrast, with multiple versions for each object, i.e., *multi-versioning* STM (MV-STM), unnecessary (or *spare*) aborts of transactions that could have been committed without violating consistency are avoided [19]. Unless a conflict between operations to access a shared object occurs, MV-STM allows the corresponding transactions to read the object's old versions, enhancing concurrency. MV-STM has been extensively studied for multiprocessors [30,31,13] and also for distributed systems [20].

In this paper, we consider scheduling transactions based on partial rollback to improve the performance in DTM. Two kinds of transactional schedulers have been studied in the past: reactive [11,2,21] and proactive [39,3]. When a conflict occurs between two transactions, the contention manager determines which transaction wins or loses, and then the loosing transaction aborts. Since

http://dx.doi.org/10.1016/j.bdr.2017.06.004
2214-5796/Published by Elsevier Inc.

aborted transactions might abort again in the future, *reactive schedulers* enqueue aborted transactions, serializing their future execution [11,2,21]. *Proactive schedulers* take a different strategy. Since it is desirable for aborted transactions to not be aborted again when re-issued, proactive schedulers abort the loosing transaction with a backoff time, which determines how long the transaction is stalled before it is re-started [39,3,20]. In the past, (reactive and proactive) transactional schedulers have been studied in multiprocessors and general distributed systems.

A distributed transaction typically has a longer execution time than a multiprocessor transaction, due to communication delays that are incurred in requesting and acquiring objects, which increases the likelihood for conflicts and thus degraded performance [3]. To boost performance in (multi-version) DTM, a transactional scheduler is therefore compelling to consider, as it effectively stalls contending transactions when conflicts occur. However, in MV DTM, reactive and proactive transactional schedulers may not be effective. First, MV-STM inherently guarantees commits of all read-only transactions [31]. Past transactional schedulers [2,21] abort loosing read-only transactions due to conflicts and simultaneously restart the aborted read-only transactions to maximize their parallelism. However, conflicts with read-only transactions do not occur in MV-STM due to multiple object versions. Thus, the parallelism of read-only transactions cannot be exploited by traditional scheduling approaches in MV-STM. Second, a transaction may request multiple objects. However, a conflict occurs and is only detected on a single object [4]. Even though other objects used by the transaction may not be subject to a conflict, the transaction is still aborted. Once a transaction is aborted, it will suffer from additional communication delays to request and retrieve all its objects again in (dataflow) DTM. Due to such delays, determining backoff times for aborted transactions (under proactive schedulers) or serializing enqueued-aborted transactions (under reactive schedulers) is generally difficult. Particularly, a long running write-intensive transaction may be easily exposed to this difficulty because of its high abort rate.

We overcome these difficulties by designing a transactional scheduler, called *partial rollback-based transactional scheduler* (or PTS). We consider MV-based transactional forwarding algorithm (MV-TFA) DTM model [20]: when transactions request and acquire an object (version), events that track the object versions are recorded. The events indicating which transaction reads or updates an object are used to detect which object is subject to the conflict. After a conflict is detected, PTS assigns different backoff times for conflicting transactions instead of aborting. If a new object version is created and conflicting transactions needing it exist, PTS sends the version to the requesting nodes. PTS identifies what write operations have caused the conflict, and the conflicting transaction is rolled-back. Thus, PTS focuses on how to protect write-intensive transactions against aborting.

We implemented PTS on Infinispan [26], and conducted comprehensive experimental studies on no and partial replications. Our studies reveal that throughput is improved by up to 2.4× over MV-TFA without PTS and 1.3× over a scalable one-copy serializable partial replication protocol (SCORe) [29] in high contention. SCORe combines a local MV concurrency control algorithm with a distributed logical clock synchronization based on Infinispan. To minimize communication delays, SCORe allows read only transactions to commit locally, but does not support any scheduler. PTS is the first ever transactional scheduler for MV-STM on in-memory data grids and constitutes the paper's contribution.

The rest of the paper is organized as follows. We overview past and related efforts in Section 2 and outline the preliminaries and the system model in Section 3. We describe PTS and analyze its properties in Section 4. Implementation and experimental studies are reported in Section 5 and conclude in Section 6.

2. Related work

Transactional scheduling has been explored in a number of multiprocessor STM efforts [12,1,39,11,2]. In [12], Dragojević et al. describe an approach that dynamically schedules transactions based on their predicted read/write access sets. In [1], Ansari et al. discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevents the two transactions from conflicting again.

Yoo and Lee present the Adaptive Transaction Scheduler (ATS) [39] that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and does not begin until dispatched by the scheduler. Dolev et al. present the CAR-STM scheduling approach [11], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queuing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Blake, Dreslinski, and Mudge propose the Proactive Transactional Scheduler in [3]. Their scheme detects hot spots of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. Evaluation on the STAMP benchmark suite [27] shows their scheduler outperforming a backoff-based policy by an average of 85%.

Attiya and Milani present the BIMODAL scheduler [2], which targets read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. BIMODAL alternates between "writing epochs" and "reading epochs" during which writing and reading transactions are given priority, respectively, ensuring greater concurrency for reading transactions. Kim and Ravindran extend the BIMODAL scheduler for DTM in [21]. Their scheduler, called Bi-interval, groups concurrent requests into read and write intervals, and exploits the tradeoff between object moving times (incurred in dataflow DTM) and concurrency of reading transactions, yielding high throughput.

Steal-On-Abort, CAR-STM, and BIMODAL enqueue aborted transactions to minimize future conflicts in SV-STM. In contrast, PTS only enqueues live transactions that conflict with other transactions. The purpose of enqueuing is to prevent contending transactions from requesting all objects again. Of course, enqueuing live transactions may lead to deadlock or livelock. Thus, PTS assigns different backoff times for each enqueued live transaction.

ATS and Proactive Transactional Scheduler determine contention intensity and use it for contention management. Unlike these schedulers which are designed for multiprocessors, predicting contention intensity will incur communication delays in distributed systems. Thus, PTS collects average running times – a history of how long transactions run – to compute a backoff time. Unlike multiprocessor STM, two communication delays will be incurred to obtain an object, one for requesting an object and the other for retrieving it. Enqueuing a live transaction during the backoff time saves those communication delays.

ATS assigns backoff times to aborted transactions. The backoff time indicates when the aborted transactions restart. If a transaction aborts, the backoff time may not be effective without considering all objects held by the transaction (if they exist). Unlike ATS, PTS gives a live transaction a backoff time indicating when the live transaction restarts if a conflict occurs.

It is important to note that, MV-STM has been extensively studied for multiprocessors and for distributed systems. MV increases concurrency by allowing transactions to read older versions of shared data, thereby minimizing conflicts and aborts. For example, Ramadan et al. present dependency-aware transactional memory (DATM) for multiprocessors [32], where transaction execution is