Enhancing locality via caching in the GMU protocol

Hugo Gomes Pimentel

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:	Professor Mário Rui Fonseca dos Santos Gomes
Supervisor:	Professor Paolo Romano
Member of the Committee:	Professor João Pedro Faria Mendonça Barreto

October 2013

Acknowlegments

I would like to start by thanking my advisor, Prof. Paolo Romano, not only for this opportunity, but also for his guidance and advices during the elaboration of the thesis.

I want to thank Pedro Ruivo and Sebastiano Peluso, for all the help they provided throughout the development of this work.

Last, but not least, I also want to thank my parents, my godmother, my brothers and my friends for their endless support, encouragement and patience.

This work has been partially supported by the project ”Cloud-TM” (co-financed by the European Commission through the contract no. 257784).

Resumo

Os protocolos de replicação parcial possuem um enorme potencial para escalabilidade, no entanto a eficiência destes sistemas pode ser fortemente prejudicada caso os padrões de acesso a dados das aplicações não exibam um bom grau de localidade. Estes sistemas normalmente recorrem a uma função de dispersão consistente para determinar a colocação dos dados, porque estas soluções permitem pesquisas locais muito eficientes e além disso, garantem que a entrada/saída de uma máquina só necessite de uma pequena mudança no mapeamento do sistema. No entanto, devido à natureza aleatória da colocação dos dados (que é independente dos padrões de acesso aos dados), também podem levar a colocações de dados sub-ótimas que prejudicam a localidade. Uma possível solução para melhorar a eficiência destes sistemas é o chamado caching, que guarda a informação que é pedida a outras máquinas para que possa ser usada futuramente e evitar novos pedidos.

Nesta dissertação, eu proponho um mecanismo de caching para o GMU [38], um protocolo de replicação parcial genuíno introduzido recentemente. Eu integrei este mecanismo numa plataforma de armazenamento de dados open-source muito popular chamada Infinispan [31], pertencente à Red Hat. Eu avalio a eficiência e a eficácia da solução apresentada através de uma extensa análise experimental, baseado tanto em ambientes de teste sintéticos como em ambientes de teste conhecidos para este tipo de sistema através da instalação dos mesmos em grandes sistemas públicos de cloud computing. Os resultados mostram que é possível obter performances 14 vezes superiores às actuais.

Keywords: replicação parcial, função de dispersão consistente, localidade, caching

Abstract

Partial replication protocols possess high potential for scalability but their actual efficiency can be seriously hindered if the applications' data access patterns do not exhibit a good degree of locality. These solutions usually adopt a data placement strategy based on consistent hashing, that allows very efficient local lookups and guarantees that the join/leave of a node incurs in a limited change in the mapping of the system. However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), it may lead to sub-optimal data placements thus hurting the locality of data. A possible approach to maximize the efficiency of these protocols is to rely on caching techniques, which replicate items frequently accessed by a node and not locally owned, in order to spare them from the costs of remote accesses.

In this dissertation, I introduce a caching protocol for GMU [38], a recent genuine partial replication protocol. I integrated the proposed caching protocol into a popular open-source transactional key-value store, namely Infinispan [31] by Red Hat. I assess the efficiency and effectiveness of the presented solution by means of an extensive experimental analysis, based on both synthetic and well known benchmarks for transactional platforms and on large scale deployments on public cloud infrastructures. The results show speed-ups up to 14 times in read dominated workloads.

Keywords: partial replication, consistent hashing, locality, caching

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The advent of the cloud computing paradigm has empowered programmers with the ability to scale out their applications easily to hundreds of nodes. However, developing applications capable of effectively exploiting the computational capabilities offered by large scale distributed cloud platforms is far from being a trivial task. This is also a consequence of the design choices characterizing many of the first generations of distributed data platforms (DTPs) for the cloud [14]. These typically maximize scalability by adopting very weak consistency models, such as eventual consistency. By relaxing consistency, these systems have been shown to achieve impressive scalability levels. However, they also shift additional burden from the platform architects to the application developers, who are exposed to the idiosyncrasies associated with concurrency and failures. Indeed, the inherent complexity of building applications on top of weakly consistent systems has been recently recognized by some of the pioneers of eventual consistency [10], and has motivated a flurry of works aimed to enforce strong consistency semantics in large scale distributed platforms [2, 12, 46, 38, 37].

By relying on multi-version concurrency control algorithms, these solutions [38, 37, 12] allow for a very efficient management of read-only transactions, sparing them from the possibility of aborting as well as from the costs of any validations. Another key property of these systems, aimed precisely to maximize their scalability is the, so called, *genuine* partial replication, according to which the execution of a transaction can only involve nodes that replicate data items it accessed [43]. This property is of the utmost importance to enable high scalability, as it rules out non-scalable solutions based either on centralized components (which may turn into bottlenecks/single point of failures) or on full-replication (which induces unacceptable overheads to propagate updates across the entire system).

1.1 Motivation

While partial replication protocols possess high potential for scalability, the actual efficiency of these systems can be seriously hindered if the applications' data access patterns do not exhibit a good degree of locality. In fact, as in these systems data is distributed across the entire set of nodes in the platform and replicated only on relatively small number of them, it follows that, in order to process a transaction's read

request, it may be necessary to fetch data remotely. This problem is particularly exacerbated since many popular key-value stores (transactional or not), such as Cassandra [29], Dynamo [14], Infinispan [31], use random placement based on consistent hashing. By relying on random hash functions to determine the location of data across nodes, these solutions allow lookups to be performed locally, in a very efficient manner [14]. Furthermore, consistent hashing guarantees that the join/leave of a node incurs in a limited change in the mapping of keys to buckets. However, due to the random nature of data placement (oblivious to the access frequencies of nodes to data), solutions based on consistent hashing may result in sub-optimal data placements. Indeed, assuming random placement of data, it is easy to see that the probability of finding a requested data item locally is inversely proportional to the number of nodes in the system. In other words, unless appropriate techniques are employed to enhance locality in the access to data, as the scale of the system grows, the number of remote read operations is destined to grow linearly, ultimately saturating the network and, consequently, hindering scalability.

A possible approach to maximize the efficiency of these protocols is to rely on caching techniques, which replicate items frequently accessed by a node and not locally owned, in order to spare them from the costs of remote accesses. However, integrating a caching mechanism in strongly consistent genuine partial replication protocols is far from being an obvious task, as it requires designing highly scalable cache consistency protocols capable of preserving transactional consistency while allowing read operations targeting data items not owned by the current node to be performed locally (based on cached data), i.e. without contacting the actual data owner.

1.2 Contributions

In this dissertation I introduce a caching protocol for GMU [38], a recent, genuine partial replication protocol that employs a fully distributed multiversioning scheme based on vector clocks. GMU has several noteworthy properties: i) it spares read-only transactions from the risk of aborts, as well as from the cost of expensive remote validations, ii) it ensures that every transaction (including update transactions that have to be aborted eventually) always observes a consistent snapshot of data, that is a snapshot producible by some linearization of a prefix of the history of committed transactions, a property called Extended Update Serializability [1] (EUS).

The presented protocol has two main innovative building blocks:

- the mechanism used to preserve the consistency guaranteed by GMU, while allowing reads targeting remote data to be served from a local cache. The proposed solution extends the vector clock-based version management and visibility logic of GMU by: i) determining a conservative upper bound on the validity of data to be inserted in the cache, and ii) exploiting this upper bound during future accesses to that cache, in order to determine whether it is safe or not for a transaction to observe cached data;

- the mechanism used to maximize the freshness of cached versions everywhere in the system, that can be used by different propagation techniques.

I integrated the proposed distributed caching scheme into a popular open-source transactional key-value store, namely Infinispan by Red Hat [31]. I assess the efficiency and effectiveness of the presented solution by means of an extensive experimental analysis, based on both synthetic and well known benchmarks for transactional platforms and on large scale deployments on public cloud infrastructures. The results speed-ups up to 14 times in read dominated workloads.

1.3 Structure of the document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 introduces GMU and the proposed caching protocol and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

Chapter 2

Related Work

2.1 Introduction

Understanding the problems addressed in this thesis implies knowing the fundamentals of different areas in distributed systems. I start this chapter by recalling the notion of atomic transaction (Section 2.2) and several alternative consistency models proposed in existing literature on transactional systems (Section 2.3). Then, I give an overview on several different classes of transactional systems (Section 2.4). Afterwards, I describe two fundamental building blocks on the construction of distributed transactional systems (which represent the focus of my thesis), namely group communication systems (Section 2.5) and replication techniques (Section 2.6). Other important features of this types of systems are data placement strategies (Section 2.7) and eviction techniques (Section 2.8). I also overview same of the basic principles of caching mechanisms (Section 2.9). Finally, I discuss more in detail specific instances of systems belonging to the categories described above (Section 2.10).

2.2 The transaction abstraction

The transaction abstraction was originally introduced in the context of DBMSs, as a way to batch multiple data manipulation operations into a single unit of work. The unit of work is treated in a coherent and reliable way by the underlying transactional system. Once started, a transaction must be ended either with commit or abort.

During its life-cycle, a transaction can pass through four distinct, well-defined states:

- *Executing*: the transaction's operations are executing;
- *Committing*: the transaction has completed the execution of its operations, and therefore, the client requested the transaction's commit;
- *Committed*: the transaction was committed;
- *Aborted*: the transaction was aborted.

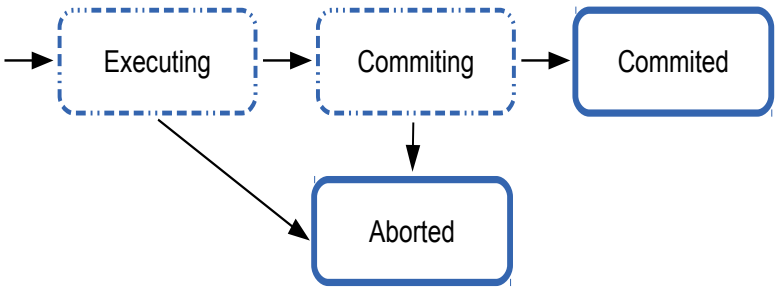

Figure 2.1: The transaction lifetime

.

Both the executing and committing states are transitory, while both the aborted and committed states are final. A committed transaction records its results in the database. A failed and then aborted transaction ends with rollback to undo its effects on the database.

Regarding properties, database transactions satisfy those commonly referred to as ACID [23]: atomicity, isolation, consistency and durability. Their description is the following:

- *Atomicity*: ensures that modifications must follow an ”all or nothing rule”, i.e., either all the modifications made by a committed transaction are made visible or none is;

- *Consistency*: ensures that each transaction changes the database from one consistent state to another consistent state;

- *Isolation*: ensures that individual memory updates within a memory transaction are hidden from concurrent transactions;

- *Durability*: ensures that once a transaction is committed, its updates will survive any subsequent malfunctions.

Transactions can also be categorized as read-only (only perform read operations) or update (perform read and/or write operations). The values obtained from read operations are stored in a structure called the read set while the values introduced by write operations are stored in a structure called the write set.

2.3 Transactional Consistency Models

2.3.1 (One Copy) Serializability

(One Copy) Serializability [5] states that the outcome of a concurrent execution of a set of transactions must be the same as the outcome of some serial execution of the same transactions. Moreover, the transactions issued by a node, which are included in this ordering, must respect the order defined in the program.

In a replicated system, a concurrent execution of transactions in multiple nodes is serializable if the outcome is equivalent to that of some serial execution of the same transactions in a single node.

2.3.2 Opacity

Opacity [21] was introduced in the context of STMs (later explained in Section 2.4.2). While the strongest coherence model of DBMSs is serializability, in STMs both serializability and opacity need to be ensured. Opacity can be viewed as an extension of serializability, with the additional requisite that even non-committed transactions are prevented from accessing inconsistent states (DBMSs only guarantee that committed transactions do not see inconsistent states), thus avoiding the occurrence of anomalies due to concurrent data accesses that can lead to arbitrary application behaviors (such as infinite loops or unhandled exceptions) and the probable crash of the entire application.

2.3.3 (Parallel) Snapshot Isolation

Snapshot Isolation [4] (SI) is a more relaxed model of consistency that allows transactions to observe a state of the system that does not necessarily reflect the updates of all transactions that have been committed. If two transactions concurrently read an overlapping set of values from the same state and make disjoint updates to different subsets of these values this will cause an ”anomaly” (with regard to the idealized system) called write skew, which never happens with serializability. However, like serializability, if the update was on same the same value, one of the transactions will be aborted.

Parallel Snapshot Isolation [46] (PSI) is a variation of this model aimed at replicated systems that allows update transactions to be applied in different orders at different nodes therefore providing better performance that SI because it does not have the cost of making update transactions be commited by the same order at different nodes.

2.3.4 (Extended) Update Serializability

Update Serializability [24] (US) is another relaxed model of consistency. US applies the same principles of serializability to update transactions. The relaxation is on the read-only transactions. Two read only transactions may read the same values but be commited by different order, a situation that does not happen in serializability.

Extended Update Serializability [1] extends the US semantic not only to transactions that commit, but to any executing transaction providing similar guarantees to the ones provided by Opacity.

2.3.5 Other models

As described in [4], there are more three models of consistency. They can be defined according to whether they suffer or not from the following phenomenons:

- Dirty Read: Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.
- Non-repeatable Read: Transaction T1 reads a data item. Another transaction T2 then modifies that data item and commits. If T1 then attempts to reread the data item, it receives a modified value.
- Phantom Read: Transaction T1 reads a set of data items. Transaction T2 then overwrites the data items that T1 has read. If T1 reads again the same data items, it gets a set of data items different from the first read.

  In the Read Uncommitted model all this phenomenons occur, while in the Read Committed model there are only Non-repeatable and Phantom Reads. Repeatable Read model only suffers from Phantom Reads. Note that in the models present above this phenomenons do not happen.

2.4 Transactional Systems

In this section, I describe several categories of systems that rely on the notion of atomic transaction. For each of these categories, I will describe in more detail specific instances that can be seen as representative of this class of transactional systems.

2.4.1 Data Base Management Systems

A Data Base Management System (DBMS) is a set of programs that enable users to store, modify, and extract information from a database. As mentioned in Section , the transaction abstraction was originally introduced in the context of DBMSs as concurrency control mechanism that allow correct execution of those operations.

DBMSs can be categorized according to the model they support or the query language or languages that are used to access the database. The relational model has been the reference model for data storage for decades and consists of three major components:

- the set of relations and set of domains that defines the way data can be represented (data structure);
- integrity rules that define the procedures to protect the data (data integrity);

- what can be done with the data (data manipulation).

A relational database supports relational algebra, consequently supporting the relational operations of the set theory. In addition to mathematical set operations namely, union, intersection, difference and Cartesian product, relational databases also support select, project, relational join and division operations. These operations are unique to relational databases.

SQL was one of the first commercial languages for this model and became the most widely used. This model has several advantages relatively to other models, namely the ease of use and flexibility. However, the expressiveness of the relation data model, as well as the richness of the data manipulations supported by SQL, make it extremely difficult to develop systems capable of scaling horizontally to a large number of nodes [33], which nowadays is very important. This has led to the development of alternative, simpler data models, which are commonly known as NoSQL, which I will describe later on Section 2.4.3.

2.4.2 Software Transactional Memories

One of the challenges of parallel programming is synchronizing concurrent access to shared memory by multiple threads. Programmers have traditionally used locks for synchronization, but lock-based synchronization has well-known pitfalls. Simplistic coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races.

Software Transactional Memory [45] (STM) was introduced to allow the programmer to compose scalable applications safely out of thread-safe libraries. As DBMS, it supports the transaction abstraction as an high-level concurrency control mechanism, so the notions of atomicity, consistency and isolation are also provided. However, unlike DMBs, STMs do not need to ensure data persistence, which allows transaction execution times typically several orders of magnitude smaller. Also, STMs can perform generic memory manipulations directly in the address space of user applications. This led to the development of new concurrency control mechanisms specifically tailored for STM environments focused on multi-core architectures, such as [8, 16, 25].

2.4.3 Distributed Transactional Platforms

With the advent of cloud computing, which is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet), there has been a proliferation of a new generation of platforms, often called Distributed Transactional Platforms (DTPs), that rely on a simpler data models, lightweight application interfaces and efficient mechanisms to achieve data durability.

The data models (key-value, column-oriented, document, graph) are often called NoSQL (Not only SQL). NoSQL is identified by the non-adherence to the use of the relational model and its query language SQL. NoSQL platforms are designed to manage large volumes of data that do not necessarily follow a fixed schema. The reduced run time flexibility compared to full SQL systems is compensated by large gains in scalability and performance.

The application of the same principles of STMs for large-scale commodity clusters (usually knowned as DSTM, i.e., Distributed Transactional Memories) is also adopted by some of this platforms because they try to leverage the fact that information stored in-memory can be retrieved much faster than from persistent storage. However, there are also new challenges posed by this non-shared memory and non-cache-coherent domain, such as communication between different machines (later described in Section 2.5), data locality management (Section 2.7), and data replication (Section 2.6).

2.5 Group Communication Systems

Group communication [11, 39] is a powerful paradigm for performing multi-point to multi-point communication by organizing nodes in groups. Typically, a system that implements this paradigm is called a Group Communication System (GCS) and offers membership and reliable broadcast services with different ordering guarantees. GCSs are used at the heart of a large plethora of distributed systems, including transactional systems. They allow programmers to concentrate on what to communicate rather than on how to communicate. Typically, broadcast services ensure all or some of the following properties:

- *Validity*: if a correct node broadcasts/multicasts a message m, then it eventually delivers m;
- *Uniform Agreement*: if a node delivers m, then all correct nodes eventually deliver m;
- *Uniform Integrity*: for any message m, every node delivers m at most once, and only if m was previously broadcasted/multicasted by its sender;
- *Uniform Total Order*: if nodes p and q both deliver messages m and m', then p delivers m before m' only if q delivers m before m'.

Uniform properties make life easier for application developers, as they apply to both correct and faulty nodes. However, enforcing uniformity often has a cost and for this reason it is important to consider whether uniformity is strictly necessary.

In the following, I overview several broadcast primitives, as these are at the basis of several transactional replication techniques, which will be the subject of Section 2.6.

2.5.1 Atomic Broadcast

Atomic Broadcast [22] (AB), also known as Total Order Broadcast, is a communication primitive that ensures that every participant receives all messages by the same order. A set of messages is broadcasted by invoking AB-broadcast and when their final order is known, they are AB-delivered. AB ensures all of the above properties.

2.5.2 Optimistic Atomic Broadcast

Optimistic Atomic Broadcast [36] (OAB) is a variant of AB that exploits the fact that in a LAN, messages normally arrive at different sites exactly in the same order. This assumption is called optimistic delivery

order. A set of messages is broadcasted by invoking AB-broadcast and OAB-delivered as soon as they arrive, but only when their final order is known, they are AB-delivered. The OAB deliver primitive enables applications to overlap computation with communication. OAB ensures all of the above properties and one more:

- *Optimistic Order*: if a node p AB-delivers message m, then p has previously OAB-delivered m.

2.5.3 Atomic Multicast

Atomic Multicast [15] (AMcast), also knowned as Total Order Multicast, is also a variant of AB. While AB sends messages to all the nodes, AMcast just sends messages to a subset of nodes. For every message m, m.dst denotes the groups to which m is multicasted. A message m is multicasted by invoking A-multicast(m) and delivered with A-deliver(m). AMcast ensures all of the above properties for m.dst.

2.6 Transactional Replication Techniques

Replication is a fundamental building block in the construction of highly available, fault-tolerant systems. Two important aspects that allow the classification of existing systems and that have a significant impact on several of their key design choices is whether data is fully or partially replicated and how they replicate the data.

In the next sections I will discuss the advantages of full and partial replication. Then I will describe some of the properties of three very known replication techniques. All of them can leverage either full or partial replication. Some of the research published so far using those techniques will also be presented.

2.6.1 Full vs Partial Replication

Full replication places copies of data items at all nodes of the system. It has the advantage of not needing expensive remote data accesses (all data is local) but is inherently unscalable, especially in presence of write-dominated workloads, because all the nodes need to be contacted upon commit.

Partial replication only assigns copies of an individual data item to a set of nodes. Ideally, when committing a transaction, the only nodes that will be contacted are the ones that maintain the data accessed by the transaction, making the system highly scalable because a node only has to execute the updates on existing data items. On the other hand, it needs to deal with expensive remote data accesses that can be very detrimental especially in presence of workloads that promote them.

Partial replication protocols can be divided in three groups [42]:

- Genuine: for each transaction T, the nodes that certificate T are the ones that have the data accessed by T;
- Quasi-genuine: for each transaction T the correct nodes that do not have data accessed by T, only store permanently the identifier of T;

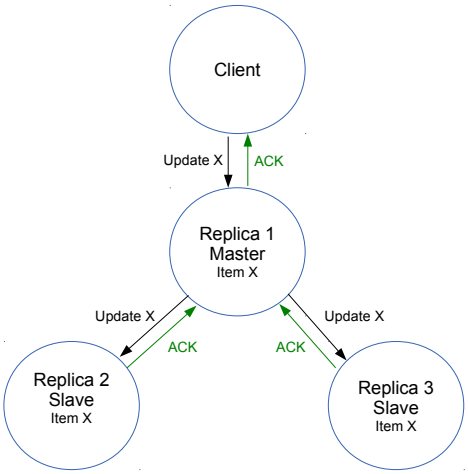

Figure 2.2: Primary-Backup in a Fully Replicated System: A Simple Execution Example

.

- Non-genuine: for each transaction T, every node stores information about T, even if they do not have data accessed by T.

Non-genuine protocols go against the goal of partial replication, because all the nodes have to be involved in a transaction, thus leading to similar problems as full replication.

2.6.2 Primary Backup

Primary backup [6], also known as master-slave or passive replication, is characterized by the existence of a node, known as the master, that nodes all requests and transfers the state updates to the remaining nodes, known as the slaves or backups. A simple execution example using this approach is depicted in Figure 2. In most cases, slaves can also nodes read-only transactions.

Primary backup often assumes the fail-stop model [7]. When the master fails one of the slaves is elected to replace it, becoming the new master. One of the problems of this technique is that in write intensive workloads the master may become a bottleneck in the system, as it is responsible for all the computation, since the slaves do not share any workload.

2.6.3 State Machine Replication

State machine replication [44], also knowned as active replication, is characterized by having all nodes processing the same sequence of requests, by the exact same order, issued by the client. To ensure the consistency of the replicated data, state machine replication requires all operations to be deterministic, otherwise the state of each node could diverge. A simple execution example is depicted in Figure 3.

Requests are processed according to the global serialization order (GSO), which is normally defined by the AB protocol used to disseminate requests or by a atomic commit protocol such as 2PC. One of the

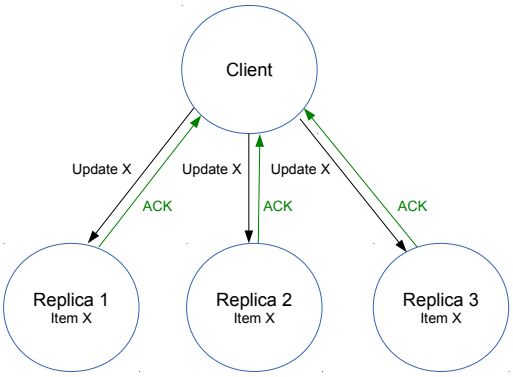

Figure 2.3: State Machine Replication in a Fully Replicated System: A Simple Execution Example

.

problems of using AB is the relatively slow communication between nodes, as it requires that consensus is reached among all nodes. Moreover, since all nodes have to execute all update requests, the ability to process them does not increase. On the contrary, it does in fact decrease, as the cost of group communication increases with the number of peers. However, as happens with passive replication, read-only requests can be processed in parallel at different nodes. In order to improve performance several state machine based schemes normally use OAB instead of AB because, as described in Section 2.5.2, OAB allows to partial overlap the transaction processing and the node coordination phases.

2.6.4 Certification Based Replication

Certification algorithms allow that the execution of a transactional request can take place at a single node, guaranteeing data coherency in the end, differently from the replication techniques described above. Specifically, these algorithms ensure that every node agree on the outcome of a transaction at commit time relying on a distributed transaction certification algorithm.

They are usually based on the deferred update model [20] and use group communication primitives. According with this model, transactions are processed locally in one node and then sent to the other nodes, at confirmation time. It was first introduced in [35], as a scheme designed to synchronize a cluster of database servers in a multi-master environment.

Existing certification-based replication algorithms can be classified into two main categories:

- *Non-voting schemes*: solutions that allow each node to certify transactions locally, by sending both the read-set and write-set via an AB primitive;
- *Voting schemes*: solutions that avoid broadcasting the read-set of transactions by sending (via AB) only the write-set.

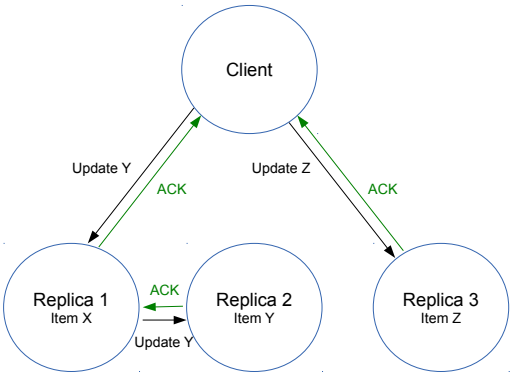

Figure 2.4: Certification Based Replication in a Partially Replicated System: A Simple Execution Example

.

Non-voting schemes are optimal in terms of communication steps but it also makes them prone to generate very large messages and to overload the network. Voting schemes drastically reduce the network bandwidth consumption but they incur into the costs of an additional coordination phase along the critical path of the transaction commit, which can reduce significantly the performance.

2.7 Data placement

In this section, I provide an overview on data placement strategies used by several transactional distributed systems.

2.7.1 Global Mapping

Global mapping is a data placement strategy that uses a data structure such as an HashMap [1] to map data items to the nodes they belong. Global mapping offers maximum opportunity for perfect resource balancing and correlation-free data placement. However, this flexibility comes at a high cost for making global decisions and consistently maintaining the map on multiple servers.

2.7.2 Consistent Hashing

Consistent Hashing [28] eliminates the cost of maintaining global maps by providing a deterministic mapping between data and nodes. There are two key issues in hashing-based schemes: (i) how to dynamically adjust the hash function in face of server failures and server additions, and (ii) how to

[1] http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

achieve load and storage balance. Consistent hashing requires minimal data relocation during server failures and expansions but it provides poor balance in data placement.

2.7.3 Grouping

Grouping is also a consistent hashing based scheme that shifts the burden of data placement to the developer. The developer has to implement an hashing scheme that groups data items by nodes. Data placement is done accordingly to where the developer wants the data to go.

2.8 Eviction

In this section, I overview some of the eviction strategies that can be used by this systems.

2.8.1 Least Recently Used

Least Recently Used (LRU) discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used item. LRU is tailored for environments where the data access patterns exclusively target the most recent items in the system.

2.8.2 Most Recently Used

Most Recently Used (MRU) discards, in contrast to LRU, the most recently used items first. It is fit for systems with random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns). MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

2.8.3 Least Frequently Used

Least Frequently Used (LFU) discards, in contrast to LRU and MRU, the less frequently used items first. LFU counts how often an item is requested and the ones used the least often are discarded first. LFU is optimal for environments where there is an access pattern that targets more frequently the same set of items.

2.8.4 Low Inter-reference Recency Set

Low Inter-reference Recency Set [27] (LIRS) is an improvement over LRU that makes eviction decisions based on the reuse distance (the number of unique memory locations referenced between a memory address' use and reuse) and recency of pages. This allows a more precise evaluation on which items to evict that only using recency as a reference.

2.9 Caching

Caching is a very general technique for improving computer system performance. Based on the principle of locality of reference, it is used in a computer's primary storage hierarchy, its operating system, networks, databases and, of course, distributed systems. Caching improves access time and reduces data traffic to data sources that have limited throughput. The work on [19] serves as reference for the arguments presented in the rest of this section.

When data is requested and can be found in the cache (cache hit), this request can be served by simply reading the cache. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which takes more time. Such mechanism must ensure that accessing cached data does not result in the violation of the application semantics, so a cache consistency maintenance algorithm is required to ensure that there is no access to stale (i.e., out-of-date) data.

Many caching algorithms have been proposed in the literature and, as all provide the same functionality, performance is a primary concern in choosing among them. They can be divided into two classes according to whether their approach to preventing stale data access is detection-based or avoidance-based. The key difference between them is that detection-based schemes are lazy, requiring transactions to check the validity of accessed data, while avoidance-based schemes are eager, as they ensure that invalid data is quickly (and atomically) removed from client caches. Both schemes allow data propagation (the newly updated value is installed at the remote site in place of the stale copy) or invalidation (removal of the stale copy from the remote cache so that it will not be accessed by any subsequent transactions).

There is no such algorithm that fits perfectly all the scenarios possible in these type of environments because there are often complex trade-offs among competing factors such as the amount of generated network traffic or the probability of transaction aborts. The avoidance/detection choice has seen to have a large impact on the number of messages sent, where detection leads to more messages.

Several of the systems already mentioned use caching schemes to improve their performance like Walter, Spanner (described in Section 2.10.7) and Infinispan (later described in Section 3.1), although applied for different purposes. Walter and Spanner use in-memory caching for fast processing of remote requests, while Infinispan uses it for enhancing data locality, which in some sense is interesting for the work that I propose to do, however it cannot be applied to GMU due to their consistency models being different.

2.9.1 Dynamic Load Management

Dynamic load management is a different strategy to enhance data locality. It differs from caching because the goal is not to store data for faster future use. Instead, the request execution or the data itself is shifted from the current place to another place that should provide better performance.

The work on [30] applies this paradigm to Massively Multiplayer Online Games (MMOGs) where servers usually employ static partitioning of their game world into distinct mini-worlds that are hosted on separate servers. They have designed and implemented an architecture that handles transient crowding

by adaptively dispersing or aggregating regions from servers in response to quality of service violations, such as the movement of many players to one area or hotspot in the game world.

Autoplacer [26] applies a similar idea to optimize data location aimed at systems that use consistent hashing as the default data placement policy by executing, cyclically, a sequence of optimization rounds. As a result of each round, a number of data items may be relocated depending if those data items pass a set of rules to determine if the expected gains of moving them are above a minimum threshold.

Tashkent+ [18] is a work focused on a load balancing technique for a replicated database that exploits knowledge of the size and contents of the working set of transactions to assign them to other nodes, focusing specifically on the ones were this transactions can be executed in-memory. This way transactions are allocated to nodes where it is more profitable their execution.

2.10 Overview of existing transactional systems

In the following, I will describe in more detail specific instances that can be seen as representative of the several building blocks described above.

2.10.1 Centralized DBMS

PostgreSQL [2], often simply Postgres, is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department [3]. Postgres pioneered many concepts that only became available in some commercial database systems much later. It supports a large part of the SQL standard and offers many modern features such as complex queries, foreign keys, triggers, updatable views, transactional integrity, multiversion concurrency control. Also, it can be extended by the user in many ways, for example by adding new data types, functions, operators, aggregate functions, index methods, and procedural languages.

2.10.2 Fully replicated and certification based DBMS

The Data Base State Machine [35] (DBSM) approach leverages the idea that all replicas receive and process the same set of inputs in exactly the same order by making the commit of a transaction a command to the replicated state-machine. More precisely, an update transaction is first executed locally at one replica. When the transaction is ready to commit, its read set and write set are broadcast to all replicas. Totally ordered broadcast is used, which means that all sites receive the same sequence of requests in the same order. When the transaction information is delivered in total order, all replicas execute a global deterministic certification procedure. The purpose of the certification is to confirm that the transaction commit respects the target database consistency criteria (typically, one-copy serializability). If the transaction passes the certification test, its write-state is applied deterministically at each replica.

[2] http://www.postgresql.org/
[3] http://www.cs.berkeley.edu/

This approach was where a certification based algorithm (later described in Section 2.6.4) was used for the first time.

2.10.3 Centralized STM

The Java Versioned STM [8] (JVSTM) is a word-based, multi-version STM that was specifically designed to optimize the execution of read-only transactions: In the JVSTM, read-only transactions have very low overheads, and never contend against any other transaction. In fact, once started, the completion of read-only transactions is wait-free in the JVSTM. To achieve this result, JVSTM uses the concept of Versioned Box (VBox) to represent transactional locations. Each VBox holds a history of values for a transactional location, by maintaining a list of bodies (VBoxBody), each with a version of the data. The access to VBoxes is always mediated by a transaction, which is created for that sole access if none is active at that moment.

2.10.4 Fully replicated and certification based DSTM

The Distributed Dependable Software Transactional Memory [13] (D2STM) is another DSTM. It provides a conventional STM interface that transparently ensures non-blocking and strong correctness guarantees (i.e., one-copy serializability) even in the presence of failures. By using JVSTM, D2STM inherits opacity and strong atomicity guarantees. It leverages a novel node synchronization scheme called Bloom Filter Certification (BFC). This is a non-voting certification scheme that exploits a Bloom filter based encoding of the transactions read-set, in order to reduce the overhead of the coordination phase. However, the probabilistic nature of the Bloom filter encoding induces false positives in the certification phase, increasing the transaction abort rate.

Speculative Certification [9] (SCert) is a fully replicated certification based replication protocol that also exploits the optimistic deliveries made by an OAB service to propagate the write-sets of committed transactions, before their GSO is established. This scheme lowers the chances of accessing a stale snapshot compared to schemes using the AB service, thus minimizing the abort rate of transactions. It also features early conflict detection, thus reducing the amount of computation and/or waiting time of transactions doomed to abort. However, (cascading) abort events are also a problem when a misspeculation occurs. SCert inhibits the creation of new transactions while the transactional state is being patched and it requires changes to already committed snapshots. This may lead to temporary inconsistencies, as these changes may not be immediately visible to concurrent threads, and to a great performance degradation, as well.

2.10.5 Fully replicated and state machine based DSTM

AGGRessively Optimistic [34] (AGGRO) is a state machine replication protocol that exploits the optimistic deliveries made by an OAB service. The key idea underlying AGGRO is the propagation of the write-sets of committing transactions to their following transactions, according to a serialization order

that is compliant with the message delivery order defined by the OAB service. When the optimistic serialization order and the GSO are not equivalent, the system triggers a (cascading) abort event for all the transactions that have directly or indirectly read from the write-set of an aborted transaction which causes a great performance degradation.

2.10.6 Partial Replicated and Certification Based DSTM

Genuine Multiversion Update Serializability [38] (GMU) is a certification based and genuine partial replication protocol for transactional systems, which exploits a distributed multi-versioning scheme that relies on a vector clock based synchronization mechanism to track both data and causal dependency relations among transactions. It guarantees EUS (Section 2.3). Unlike previous multi-version based solutions, GMU does not rely on a global logical clock, which represents a contention point and can limit system scalability. Also, GMU never aborts read-only transactions and spares them from distributed validation schemes, which is a major source of inefficiency on other systems. However, when scaling to a large number of nodes, the overhead of the messages exchanged increases, because the size of the vector clocks also increases. GMU is the focus of the work presented in this report and will be explained in more detail in Section 3.2.

SCore [37] is another certification based and genuine partial replication protocol for transactional systems. Like GMU, it leverages a multi-version concurrency control algorithm and never aborts read-only transactions and spares them from distributed validation schemes. However, unlike GMU, SCore uses a distributed logical-clock synchronization scheme that only requires the exchange of a scalar clock value among the nodes involved in the handling of a transaction thus providing less overhead than a vector clock based solution. Also, unlike GMU, SCore provides 1CS (Section 2.1).

2.10.7 Geo-Replicated Systems

Walter [46] is a partial geo-replicated and certification based DSTM that provides the PSI (Section 2.3.3) consistency model. It introduces two novel mechanisms that are also used to implement PSI: preferred sites and counting sets. Each object has a preferred site that corresponds to the datacenter closer to the owner of that object. This enables transactions that are local to the preferred site to be committed more efficiently using a Fast Commit protocol. This protocol allows for a transaction to commit at the preferred site without contacting other sites. However, if the transaction is not local it must execute a Slow Commit protocol. This protocol consists in a two-phase commit protocol between the preferred sites of the objects being written. The counting sets are a new data type, similar to commutative replicated data types , that allows to avoid write-write conflicts. Operations where counting sets are acceptable can be quickly committed using the Fast Commit protocol, improving the throughput in those cases. However, the fact that transactions are executed and committed by a central server at each site corresponds to a bottleneck in performance. Moreover, if transactions are not local, then they must be committed by the Slow Commit protocol, which can limit the throughput and scalability.

Google has recently introduced Spanner [12], a fully geo-replicated datastore. This system has

evolved from Bigtable [10] and has the purpose of covering the flaws of Megastore [2], which provides low write performance over the wide-area. Spanner shards data across many nodes all over the world in order to provide global availability and geographic locality to clients. The data shards are maintained in many Paxos state machines that are responsible for guaranteeing the consistency of the data. This system also provides linearizable transactions by using globally-meaningful commit timestamps that guarantee the total order of operations. These timestamps are based on uncertainty bounds and are assigned by a service called TrueTime, which makes use of GPS and atomic clocks as reference.

Chapter 3

The GMU protocol

In this chapter, I present an overview of the baseline protocol for the proposed caching mechanism, GMU (Section 3.2), but without first introducing the distributed transactional platform behind it, Infinispan (Section 3.1) and the model of the target system (Section 3.2).

3.1 Infinispan

Infinispan [31] by RedHat, is a highly scalable NoSQL data transactional platform that maintains data entirely in-memory relying on replication as its primary mechanism to ensure fault-tolerance and data durability. It exposes a key-value store data model and supports both partial and full data replication.

As other recent NoSQL platforms, Infinispan opts for weakening consistency in order to maximize performance. It uses the classical 2PC algorithm to maintain data coherence and provides weak consistency guarantees: Read Committed (RC) and Repeatable Read (RR) (Section 2.2.4). However, as mentioned in the introduction of this work, the inherent complexity of building applications on top of weakly consistent systems is a big limitation in this systems and this is one of the reasons I choose GMU as the base protocol for the caching mechanism.

In Infinispan architecture, each one of its nodes is composed by the following components:

- *Transaction Manager*: this component is responsible for the execution of transactions, either local or remote;
- *Lock Manager*: this component is responsible for managing the locks acquired by the transactions and is also able to detect distributed deadlocks. If there is a distributed deadlock between two transactions, one of them will be canceled;
- *Replication Manager*: this component is responsible for the maintenance of data coherence between nodes. It certifies transactions through the 2PC algorithm and the Transaction Manager;
- *Persistent Storage*: this component is responsible for guaranteeing that the storing and loading of data in a persistent manner (in a disk or a DB, local or remote). If this component is active, it can work on one of the following execution modes:

- *Activation/Passivation*: the data is stored persistently or in memory. When a item of that data is needed, it is moved to the memory (and is deleted from the disk/DB). When it is needed no more, that item is deleted from memory and stored again persistently;
- *Load/Store*: the data is stored in both memory and persistently. When a data item is needed, a copy of that item is sent to memory.

– *JGroups*: the GCS of Infinispan that is responsible for the maintenance of group members information (including fault detection) and for the support of the communication between nodes.

In the context of the Cloud-TM Project[1], Infinispan has been extended with the GMU protocol, which is described in detail in Section 3.3.

3.2 Model of the target system

I consider a classic asynchronous distributed system model composed of $\Pi = \{p_1, \ldots, p_n\}$ nodes. Nodes communicate through message passing and do not have access to a shared memory nor a global clock. Messages may experience arbitrarily long (but finite) delays and I assume no bound on relative site speeds or clock skews. I consider the classic crash-stop failure model: sites may fail by crashing, but do not behave maliciously. A site that never crashes is correct; otherwise it is faulty.

Each node p_i stores a partial copy of data, for which I assume a simple key-value model. Each data item d is a sequence of versions $\langle k, val, ver \rangle$, where k is a key representing d's identifier, val is its value and ver is a scalar, monotonically increasing logical timestamp that identifies (and totally orders) the versions of a data item d. For the sake of brevity, I will use the notation v to denote the v-th version of the value associated with key k.

I abstract over the data placement policy by assuming that data is subdivided across m partitions, and that each partition is replicated across r nodes (in other words, r represents the replication degree for each data item). I denote with $\Gamma = \{g_1, \ldots, g_m\}$ the set of groups of nodes g_j that replicate the j-th data partition. Each group is composed of exactly r nodes (to ensure the target replication degree), of which at least one is assumed to be correct. In order to maximize flexibility of the data placement strategy, I do not require groups to be disjoint (they can have nodes in common), and assume that a node may participate to multiple groups, as long as $\bigcup_{j=1\ldots m} g_j = \Pi$. I denote with $groups(p_i)$ the set of groups to which p_i belongs, and with $proc(g_j)$ the set of node that replicate data belonging to partition j.

I model transactions as a sequence of read and write operations on data items, preceded by a begin, and followed by a commit or abort operation. Transactions originate on a node $p_i \in \Pi$, and can read/write data belonging to any partition. Also, I do not assume any a-priori knowledge on the set of data items read or written by transactions. Given a data item d, I denote as $owners(d)$ the set of nodes that maintain a replica of d (namely the nodes of the group g_j that replicate the data partition containing d).

[1] http://www.cloudtm.eu/

3.3 Overview of the GMU protocol

As classical multiversion concurrency control schemes, GMU maintains a chain of totally ordered committed versions for each data item on every node and the versions are associated to scalar version numbers.

The order is determined by the version numbers that follow the order of commits on a given node hence determining a relationship between a snapshot locally committed on a node p_i and all the versions written with that commit on p_i. A snapshot committed on multiple nodes is globally identified by a vector clock (with size equals to the cardinality of Π) that univocally determines the versions written in that snapshot and keeps track of the dependencies with the other commits on the involved nodes. In addition each node p_i maintains (i) a scalar logical clock, i.e., $LastPrepSC$, used during the commit of a transaction to assemble its commit vector clock and (ii) the history of the committed vector clocks on p_i stored in a list named $CLog$ and ordered in accordance with the order of commits on p_i.

During execution, a transaction T maintains (i) a reading vector clock, i.e. $T.VC$, used as a visibility reference during read operations, that keeps track of the causal and data dependencies created by T during its execution and (ii) a vector of boolean values, i.e. $T.hasRead$, that maintains at the $i\text{-}th$ entry the information on whether the current transaction has read on the node p_i or not.

Read operations require the determination of which version among the versions maintained by the data platform should be visible to the transaction. In general, GMU determines which version of a item should be returned upon the execution of a read operation on node p_i by transaction T according to the following rules:

- Rule 1 - *Reading Lower Bound*: as data is replicated among multiple nodes, it is possible that node N may have not yet finalized the commit of a transaction $T*$ whose effects have been already observed by transaction T (during a previous operation) on another node. In order to avoid consistency issues, because the vector clock limiting the visibility range cannot precede causally the current $T.VC$ reading vector clock, T is blocked until the N-th entry of T's vector clock is larger than the $i\text{-}th$ entry of the most recent vector clock in the $cLog$ of p_i;
- Rule 2 - *Reading Upper Bound*: in order to maximize data freshness, if transaction T is reading for the first time on node p_i, the vector clock of T is updated in its $i\text{-}th$ entry with the $i\text{-}th$ entry of the vector clock retrieved from the read. However, during subsequent read operations the $i\text{-}th$ entry of the vector clock of T cannot be advanced anymore, meaning that GMU establishes an upper bound on the freshness of the snapshot that can be observed by T, that avoids the reading of versions created by transactions serialized after T;
- Rule 3 - *Data Version Selection*: as in a classic, non-distributed multi-version concurrency control schemes [5], whenever there are multiple entries of some data item, the entry selected will be the most recent one that has a vector clock smaller or equal than the vector clock of T.

Figure 3.5. and 3.6. depict two different scenarios where the reading rules are applied. Starting with the scenario in Figure 3.5., it represents a system with three nodes where Node 1 and Node 2

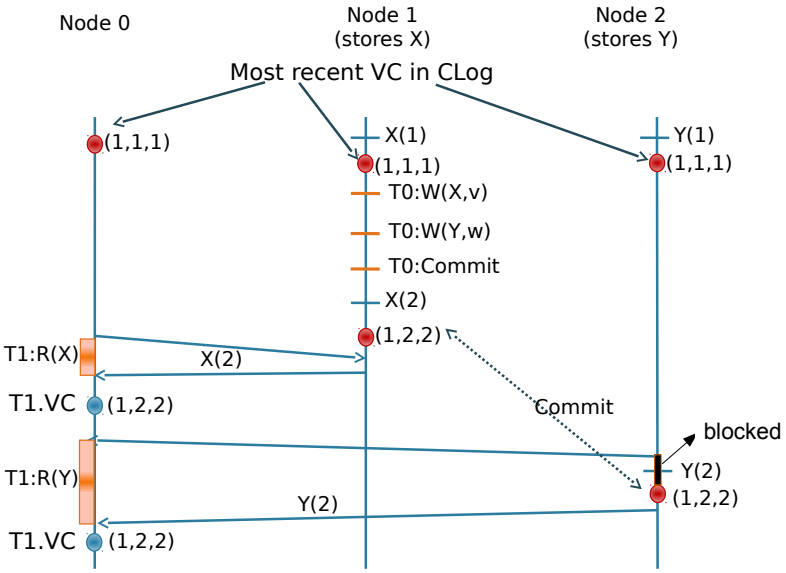

Figure 3.1: Read Operations in GMU: Application of the Reading Rules

.

store items X and Y, respectively. All the nodes have the most recent vector clock in their $cLog$ equal to (1,1,1), when transaction T0 starts its execution on Node 1 with T:VC (1,1,1) and commits with T.VC equal to (1,2,2), creating a new version of both X and Y. The logic behind the execution of the commit phase of a transaction will be explained in detail later, because in this case I just want to show the use of the reading rules. After the commit of T0 in Node 1, T1 starts its execution on Node 0 where its first operation is to read X from Node 1. Applying Rule 2, after reading X(2) from Node 1, T.VC will become (1,2,2). More in detail, on the first read operation on Node 1, the $T.VC$ and the $T.hasRead$ vectors are used to determine in the $cLog$ the freshest committed vector clock $MaxVC$ whose associated snapshot can be still readable by T in accordance with its past history and without incurring in a violation of the correctness criteria. This property guarantees that read-only transactions are always correctly serialized within histories of committed update transactions, without the need for incurring in expensive distributed validations or aborts (thanks to the multi-version scheme, read-only transactions are allowed to observe previously committed versions of data). Going back to the scenario, after the reading of X(2) by T1 there is a read for Y that belongs to Node 2. However, because the commit of T0 takes longer in Node 2 than in Node 1, the new version of Y is not visible yet for T1. Applying Rule 1, T1 waits for the commit of T0 on Node 2 and only after that returns the new version of Y, Y(2). In both reads Rule 3 is also applied because X and Y are the most recent versions in both nodes and are visible for T0.

Figure 3.6. shows a different application of the rules. In this case the configuration is similar to the one in Scenario 1, however T0 starts in Node 0 with T.VC equal to (1,1,1) and wants to read X. Applying Rule 2, when reading X, T.VC will stay the same because the retrieved version, X(1), is the most recent version of X in Node 1 and the most recent vector clock in the $cLog$ is (1,1,1). After the reading of X, T0

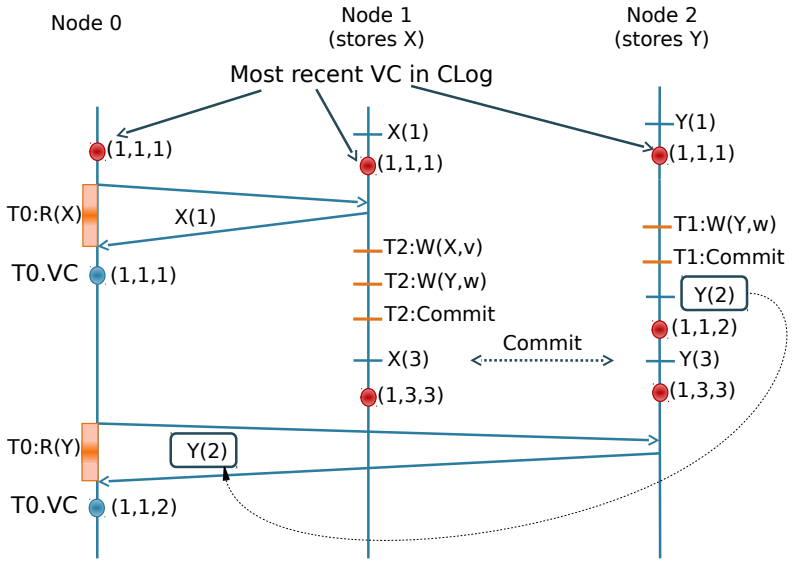

Figure 3.2: Read Operations in GMU: Application of the Reading Rules 2

.

will read Y, however, in the meanwhile, T2 starts its execution and commits new version of X and Y, X(3) and Y(3), respectively. The most recent vector clock in the $cLog$ of Node 1 and 2 becomes (1,3,3). When T0 reads Y, the retrieved version is Y(2) and not Y(3) because applying Rule 3, the most recent version of Y for T according to its vector clock is Y(2). Regarding Rule 1, because there are no dependencies between transactions, there is no need to block them until the dependencies are solved.

Moving on to the write operations, when an update transaction T executes, it only stores the identifier and the version to be written in its write-set. However, when T commits, it is assigned a commit vector clock $commitVC$ that is the outcome of a genuine consensus phase executed by only the nodes that replicate items read or to be written by T, i.e. the participants. $commitVC$ represents the identifier of the new snapshot created by T and defines a serialization order of T among the committed update transactions that conflict with T. The consensus is reached by means of a Two-Phase Commit run in which the coordinator, i.e. the node that executes T, computes the $commitVC$ combining the proposals sent by the participants upon the reception of a prepare message from the coordinator. Then it sends the $commitVC$ to all the participants via a commit message. Each participant computes its proposal by combining the information in its $CLog$ and $LastPrepSC$, but only after having (i) acquired shared locks on T's read-set and exclusive locks on T's write-set and (ii) validated T's read-set; afterwards upon the $T.commitVC$ is received, T is inserted in a queue that contains pending transactions waiting for the commit and ordered according to their $commitVC$. The actual commit of a transaction is executed according to the order defined in the queue and entails (i) adding the associated $commitVC$ in the $CLog$ and (ii) writing a new version for each key in the write-set by associating a version number derived from $commitVC$. When the validation of T fails, T is aborted. Also, since update transactions need to read

the most recent version of the data items, when this does not happen, the transaction is early aborted, i.e., aborted on the execution phase.

Chapter 4

Caching in GMU protocol

In this chapter, I introduce the proposed caching mechanism (Section 3.1) by explaining in detail the design choices made and by providing some examples on why those choices were necessary.

4.1 The caching protocol

There are two main problems that need to be addressed while designing a caching mechanism:

- it is necessary to ensure that accessing cached data does not result in the violation of the correctness properties;
- it is important that the employment of the caching algorithm does not have a negative impact on the freshness of the data observed by transactions, or it may end up hindering performance, rather than improving it.

In the following, I discuss how I plan to solve this problems specifically for GMU by presenting:

- a cache consistency algorithm, that allows to determine whether a transaction can safely read a cached version of data, i.e. without compromising the consistency criterion provided by GMU (namely Extended Update Serializability [1], see Section 2.2.3);
- an invalidation scheme that can use different dissemination techniques aimed at maximizing the freshness of the data maintained in cache and, hence, at enhancing the profitability of the caching mechanism.

  I also provide a discussion on why reading from the caching mechanism is similar to reading in a remote node (Section 4.2.2).

4.1.1 Ensuring data consistency

To ensure that data is safely read, the cached data is maintained, analogously to non cached data in GMU, in a multi-version data container. However, unlike the non cached data container, each data

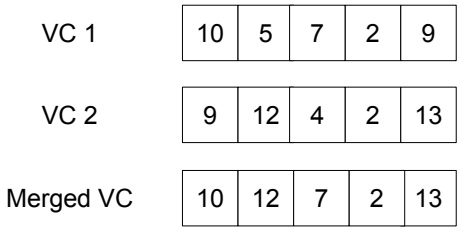

Figure 4.1: Merging and maximizing two VCs: A simple example

.

item d, is a sequence of versions $\langle k, val, creationVC, validityVC \rangle$, where k and val identify exactly the same as before and $creationVC$ is the vector clock of the transaction that originated this version, and $validityVC$ is a vector clock that represents up to when this version is valid.

As it will be discussed in the following, by using these two pieces of information, it is possible to implement the same set of rules determining version visibility as in GMU without compromising consistency.

The pseudo code describing the behavior of a read from the cached data container is reported in Algorithm 1. When a transaction T needs to read a data item d that is not local, with the introduction of the caching mechanism, now T first searches the cached data container for d and if d is not found, than T issues a remote request for it. Specifically, when searching the cached data container, first (line 4) T checks if d exists. If it exists, then to use d it needs to make sure that d has a version v that was created before T. This is simple done by checking the $creationVC$ of v and the vector clock of T and if T has already read from some node in the system (line 5). If T has not read from any node (this is the first read of T), T can read v (lines 7-9). If not, it is necessary to compare the entries in $creationVC$ where T has already read from with T's vector clock on the same entries (lines 11-12). In case this check fails, an older version will be checked, in case there are older versions (lines 13-14), because versions are ordered by their $creationVC$. When v is found, additional checks are performed to determine whether it is safe for T to read v. First, T gets the entry of the owner of v, lets say j, in the $validityVC$ and compares it to T's vector clock in the j-th position (line 8). If this check succeeds, it means that v is valid for T and can read by it, but T still needs to compare the entries in the $validityVC$ from where T has already read from with T's vector clock in the same entries (lines 27-31). This check is done to make sure that the reading Rule 2 of GMU (described in Section 3.3.) is not violated, i.e., to avoid that T observes a too fresh snapshot and violates consistency. If it succeeds, the merged and

Algorithm 1: Read operations on local node

```
1  [Key, Ver, VC] getValidVersion(Key k, VC T.VC, bool[] hasRead)
2  Versions vers ← getVersions(k);
3  if vers ≠ null then
4      Ver v ← findVisibleVersion(vers, T.VC, hasRead);
5      if v ≠ null then
6          VC validityVC ← v.validityVC;
7          if validityVC[j] ≥ T.VC[j] then // p_j = owner(k)
8              VC updatedVC;
9              if checkMaxVCCondition(validityVC, T.VC, hasRead) then
10                 updatedVC ← mergeAndMax(validityVC, T.VC);
11             else
12                 updatedVC ← mergeAndMax(v.creationVC, T.VC);
13             return [key, v, updatedVC];
14 return null;

15 Ver findVisibleVersion(Versions vers, VC T.VC, bool[] hasRead)
16 Ver v ← vers.mostRecent;
17 if ∄j: hasRead[j] = true then
18     return v;
19 while v ≠ null do
20     if ∀j: hasRead[j] = true ∧ v.creationVC[j] ≤ T.VC[j] then
21         return v;
22     else
23         v ← v.prev;
24 return null;

25 bool checkMaxVCCondition(VC validityVC, VC T.VC, bool[] hasRead)
26 if ∄j: hasRead[j] = true then
27     return true;
28 if ∃j: validityVC[j] > T.VC[j] ∧ hasRead[j] = true then
29     return false;
30 return true;

31 VC mergeAndMax(VC vc1, VC vc2)
32 VC mergedVC ← VC.newVC;
33 for i = 0 to vc1.size do
34     if vc1[i] > vc2[i] then
35         mergedVC[i] ← vc1[i];
36     else
37         mergedVC[i] ← vc2[i];
38 return mergedVC;
```

maximized vector clock (lines 31-38 and depicted in Figure 4.7.) between the $validityVC$ and T's vector clock becomes the new T's vector clock. If not, the new T's vector clock is the merged and maximized vector clock between the $creationVC$ and T's vector clock. Going back to line 8, if the check fails, v is considered too old because it may exist a new version on the remote node that it is not known yet in the current node. When some of the checks do not succeed (lines 4, 6, 8), a cache miss is forced and the remote request for d is executed.

Algorithm 2: Read operations on remote node (node p_j)

```
1  on receive READREQ[Key k, VC T.VC, bool[] T.hasRead] from p_i
2  [Ver readV, Ver nextV, VC updatedXactVC, bool last] ← GMURead(k, T.VC, T.hasRead);
3  VC creationVC ← getCreationVC(readV);
4  VC validityVC] ← getValidityVC(readV, nextV);
5  send [readV, updatedXactVC, validityVC, creationVC, last] to p_i

6  VC getValidityVC(Ver readV, Ver nextV)
7  if nextV = null then
8      return CLog.mostRecentVC;
9  else
10     foreach vc ∈ CLog do
11         if vc[j] < nextV.value then
12             return vc;
13 return null;

14 VC getCreationVC(Ver readV)
15 foreach vc ∈ CLog do
16     if vc[j] = readV.value then
17         return vc;
18 return null;
```

Algorithm 2 describes the remote operations introduced with the caching mechanism to retrieve the $creationVC$ and $validityVC$ from a remote node. As already explained, when a node p_i needs a data item d owned by node p_j, it issues a remote request to p_j. In addition to the normal execution of this remote request (line 2), it also always gathers the $creationVC$ and $valitityVC$ of the version v returned by the remote node(lines 3-4).

Since v is known, getting v's $creationVC$ is simply done by searching in the $cLog$ for the transaction that created v (lines 15-18).The $validityVC$ is calculated by using either i) the most recent entry in $CLog$, if the v is the freshest one (lines 7-8), or ii) the vector clock that identifies the snapshot existing on p_j before the commit of the transaction that overwrote v (lines 9-12).

4.1.2 Why the algorithm is necessary?

In this section, I will describe two scenarios (depicted in Figure 4.8 and 4.9) that will help understand why reading from the cache using the cache consistency algorithm presented above does not lead to a violation of the Reading Rules of the GMU protocol (see Section 3.3). Starting by Figure 4.8, it shows an

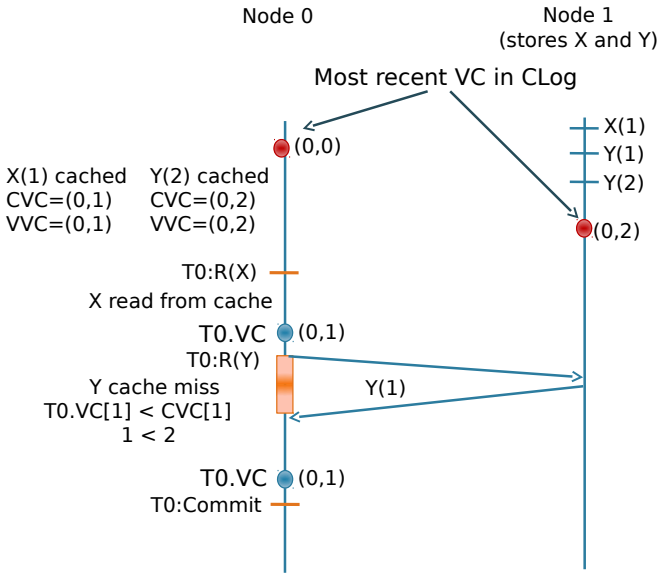

Figure 4.2: The caching mechanism: A cache miss example

.

execution example where line 4 and subsequently line 5 of Algorithm 1 are needed to maintain reading consistency. In this scenario, there are two nodes, Node 0 and Node 1, where items X and Y belong to Node 1 and were previously written by some transactions that created versions X(1), Y(1) and Y(2), but are not important for the goal of this demonstration. Version X(1) is stored in the cache of Node 0 with $creationVC$ equal to (0,1) and $validityVC$ equal to (0,1) and Y(2) with $creationVC$ equal to (0,2) and $validityVC$ equal to (0,2). Note that Y(1) is not cached. The most recent vector clock in the $cLog$ of Node 0 is equal to (0,0) and in Node 1 is equal to (0,2). T0 starts its execution in Node 0 with T.VC equal to (0,0), by reading X. Since X(1) is in the cache, T0 will try to read it and, algorithm aside, it can and would not cause an inconsistency, because looking at Node 1 there is only X(1) there. After that, T0 reads Y and because there is Y(2) in cache, T0 will try to read it too. However,if T0 was allowed to observe Y(2) from the cache, this would cause the violation of Rules 2 and 3 because Y(2) was serialized after T0 and is not the most recent version visible by T0. By applying lines 4-5, T0 will force a cache miss and retrieve Y(1) from Node 1.

Figure 4.9 shows an execution example where line 7 of Algorithm 1 is also needed to maintain reading consistency. In this scenario, there are also two nodes, Node 0 and 1, however item X belongs to Node 0 and item Y to Node 1. Also, they were previously written by some transaction that created versions X(1) and Y(1). Y(1) is cache in Node 0 with $creationVC$ equal to (1,1) and $validityVC$ equal to (1,1). The most recent vector clock in the $cLog$ of Node 0 and Node 1 is equal (1,1). T0 starts its execution in Node 1 and commits two new versions of X and Y, X(2) and Y(2), respectively, making the most recent recent vector in the $cLog$ of both nodes equal to (2,2). The most recent vector clock in the $cLog$ of Node 0 is (0,0) and in Node 1 is (0,2). T1 starts its execution on Node 0 with T.VC equal to (2,2)

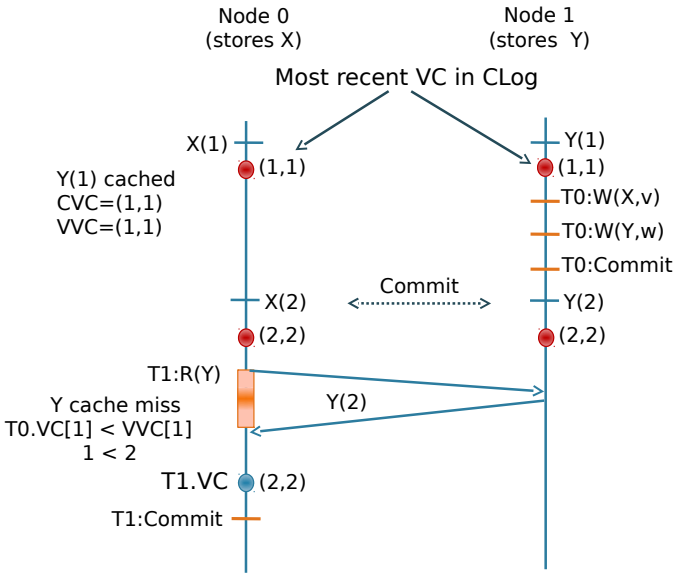

Figure 4.3: The caching mechanism: A cache miss example 2

.

by reading Y. If T1 was allowed to access the Y(1) in cache it would cause the violation of Rule 1 and 3 because T0 was serialized after Y(1) and Y(1) is not the most recent version that is visible by T0. By applying line 7, T1 will force a cache miss and retrieve Y(2) from Node 1. In lines 8-13, Rule 2 is also enforced.

4.1.3 Maximizing data freshness

The employment of the caching algorithm should also not have a negative impact on the freshness of the data observed by transactions, or it may end up hindering performance, rather than improving it. Using just the algorithm I described so far is not enough to achieve this goal. Going back to the scenario depicted in Figure 4.9., because of the commit of transaction T0 that touches both nodes, the version of Y that is in cache in Node 0 is no longer visible when T1 starts. Now imagine that instead of having just Y, Node 0 had 50000 data items that belong to Node 1 with the same $validityVC$ as Y. This would mean that all those 50000 data items would be no longer visible after the increase of the vector clock attributed to transactions upon their start (as a consequence of the commit of T0) and the only way to make them visible again would be doing remote requests to all of them. This is a big problem in systems with a very large number of data items and with a lot of transactions that touch more than one node, because the caching mechanism would not be exploited effectively. Worse, the performance of the system would be even worse than in a system not using caching, because now it also incur in overheads for checking and enforcing the validity of cached data.

To solve this problem, I designed an additional protocol that allows the caches of each node to

invalidate versions that were overwritten and advance the $validityVC$s of all the other ones, making them the freshest possible. Note that the term invalidation does not mean the removal of those versions. In this case, my goal is to make those versions maintain the same $validityVC$ while advancing the others.

Algorithm 3: Invalidation operations on sender node(node p_i)

```
1  [long, Set, VC] getInvalidationSet(ID j)
2  VC mostRecentVC ← CLog.mostRecentVC;
3  long lastSentValue ← CLog.lastSentValues[j];
4  long mostRecentValue ← mostRecentVC[i];
5  Set iSet ← ∅;
6  if mostRecentValue > lastSentValue then
7      if compareAndSet(CLog.lastSentValues[j], lastSentValue, mostRecentValue) then
8          foreach vc ∈ CLog do
9              if vc[i] > lastSentValue then
10                 foreach w ∈ vc.keysCommitted do
11                     if isPrimaryOwner(i, w) then
12                         iSet.add(w);
13         return [lastSentValue, iSet, mostRecentVC];
14 return null;
```

Algorithm 3 describes how this is done. The idea is to gather what I named an invalidation set (iSet), i.e., a set of keys to invalidate (lines 8-12) and the most recent vector clock on the $cLog$ (line 2). But because the $cLog$ is constantly changing we do not need to gather all the keys there, just the ones from the most recent entry up to the last ones that were gathered before (line 9). Furthermore, there are two situations in which it is possible to try to gather the iSet but it returns nothing. First (line 6), it can happen that there is nothing to gather. On the other hand, if there is, when more than one thread tries to gather an iSet, only one thread will get it (line 7), because there is no need to have duplicate iSets.

Having the scheme to invalidate keys is not enough because this information needs to be sent to the other nodes. I implemented three different dissemination strategies, named and described in the following:

- EAGER: an iSet is disseminated everytime a transaction is committed at some node to all the other nodes even if the node has no keys belonging to the sender node;
- BATCH: an iSet is disseminated at a fixed time based rate to all the other nodes even if a node has no keys belonging to the sender node;
- LAZY: an iSet is only disseminated when a node receives a remote request from other node. The iSet is then piggybacked in the remote request response.

I evaluate the effectiveness of those strategies in Chapter 5.

Other important important aspect to consider when using the invalidation mechanism is its efficiency. As already mentioned, these systems usually stored a large number of keys, so, when a node receives an iSet to process, invalidating the keys present in the iSet should be a fast process because the amount

Algorithm 4: Invalidation operations on receiver node(node p_j)

```
1  void invalidateKeys(ID i, Set iSet, VC mostRecentVC)
2  foreach invalidKey ∈ iSet do
3      Versions vers ← getVersions(invalidKey);
4      if vers ≠ null then
5          Ver v ← vers.mostRecent;
6          if v.validity.isShared() then
7              v.validity ← [v.validity.VC, false];
8  Validity mostRecentValidity ← mostRecentValidities.get(i);
9  if mostRecentValidity ≠ null then
10     Validity validity ← [mostRecentVC, true];
11     mostRecentValidities.put(i,validity);
12 else
13     mostRecentValidity.VC ← mostRecentVC;
```

of keys is small in most cases. However, because there is a need to advance the $validityVC$s of all the other keys belonging to that node, updating each key by itself will be a major source of inefficiency. To solve this problem, I developed a scheme that groups keys that belong to some node with a common shared $validityVC$. Specifically, I added a new data structure called $Validity$ that contains the $validityVC$ and a flag $shared$ to better control the linking and unlinking of VCs. Each node will stored a number of shared $validity$ values equal to all the other nodes in the system. Because all the nodes are primary owners of a disjoint set of keys, it is possible to link those keys to the $valitity$ value of each node. The linking and unlinking process is done in the invalidation mechanism but also when storing a new item in the cache to make sure that older versions do not remain linked to their shared $validity$ value.

Algorithm 4 shows how the invalidation process works. First, the most recent version (line 5) of each key that is present in the iSet (line 2) and on the cache (3-4) is invalidated if it has a shared $validity$ value (lines 6-7). Then the shared $validity$ value is created if it does not exist (lines 8-11), otherwise its $validityVC$ is advanced (lines 12-13). In in this case, there is only the need to update the shared $validity$ value and all the keys linked to it are automatically updated.

4.1.4 Freshness of the initial transaction vector clock

In normal GMU, when a transaction starts on a node its first vector clock is the most recent one in $cLog$. However, because I introduced the caching mechanism with the invalidation scheme, each node has now a global vision on what are the most recent committed vector clocks in all the other nodes because of the $validity$ values. This means that using those values, transactions can start with fresher vector clock than the most recent one in the local $cLog$. For read-only transactions this only matters if freshness is an important requisite of the system, because they can be, thanks to multi-versioning, serialized in the past. However, update transactions are forced to read the most recent version of the data items they will update, or they will abort. There is a critical situation that can happen when caching a version that is not the most recent one. Imagine a scenario where some update transaction T originated on Node N

with the most recent vector clock in $cLog$ equal to mrv, reads for the first time item X that has a cached version X(1) that when was retrieved was not the most recent one,i.e., X(2) already existed but was not visible by the transaction. Assuming that X(1) is visible by T, T will early abort because thats the rule in GMU (Section 3.2). In a workload were aborted transactions are repeated until they succeed, this situation will lead to an spiral of aborts and will not let the workload resume its normal execution mainly because the initial transaction vector clock will always be mrv until some other transaction commits on N and changes it. However, if instead of using mrv, T uses a vector clock equal to $gmrv$, that is the freshest global vector clock calculated using the $validity$ values, this situation does not happen, because X(2) will now be visible by T.

Algorithm 5: Transaction initialization on local node(node p_i)

```
1  void initLocalTransaction(LocalTransaction tx)
2  VC gmrv ← computeGlobalMostRecentVC();
3  VC mrv ← CLog.mostRecentVC;
4  if gmrv = null then
5      tx.VC ← mrv;
6  else
7      tx.VC ← mergeAndMax(mrv, gmrv);

8  VC computeGlobalMostRecentVC()
9  VC gmrv ← VC.newVC;
10 foreach validity ∈ mostRecentValidities do
11     gmrv ← mergeAndMax(gmrv,validity.VC);
12 return gmrv;
```

Algorithm 5 exploits precisely this idea. Instead of starting the transaction vector clock with mrv, that probably did not see most of the commits on other nodes, thus being less fresh in a global sense, it is easy to used all those $validity$ values gathered by the invalidation scheme and merge and maximize them to $gmrv$. If $gmrv$ is not present (line 4-5), because the node has not received invalidation messages from the other nodes, the transaction starts exactly like it did before. If it is (lines 6-7), the new transaction will get the vector clock created by the merge and maximize of mrv with $gmrv$.

Chapter 5

Evaluation

In this chapter, I provide an experimental study of the proposed caching mechanism for GMU that, as mentioned before, was integrated into Infinispan, a mainstream in-memory distributed transactional platform developed by Red Hat. First, I present the goals of this study (Section 5.1). After that, I overview the environments were the experimental study was conducted (Section 5.2) and the workload configurations used (Section 5.3). Then, I present each benchmark by describing their main properties and the results obtained (Sections 5.4, 5.5, 5.6). At last, I wrap up this section by providing a discussion of the pros and cons of the caching mechanism using different invalidation strategies (Section 5.7).

5.1 Goal

With this experimental study I aim to answer the following questions:

- How much can the caching protocol enhance GMU scalability in different scenarios?
- How much impact it has on the abort rate?
- How high is the cache hit percentage in different scenarios?
- How much overhead does the caching protocol introduce?

5.2 Environments

In the following, I provide a description of the environments where this experimental study was conducted.

5.2.1 CloudTM

CloudTM is an OpenStack-based cloud computing infrastructure, deployed in a dedicated cluster. Each machine is equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors, 40 GB of RAM and interconnected via a private Gigabit Ethernet. The VMs instantiated via OpenStack were allocated

1 physical core plus 4GB Ram and the virtualization took advantage of the hardware support provided by the Intel(R) processors.

5.2.2 FutureGrid

FutureGrid [1] is a public distributed test-bed for parallel and cloud computing. This platform allows us to an evaluation in environments representative of public cloud infrastructures, which are typically characterized by more competitive resource sharing, ample usage of virtualization technology, and relatively less powerful nodes. In the FutureGrid platform it was performed experiments using 16 to 80 virtual machines, equipped with 4GB RAM, one 2.93GHz core Intel Xeon CPU X5570, running CentOS 5.5 x86_64. All the VMs were deployed in the same physical data-center and interconnected via InfiniBand, a switched fabric computer network communications link used in high-performance computing and enterprise data centers.

5.3 Configuration

For all the benchmarks runned I used two workloads:

- Workload A - a read-only dominated workload with 90% read-only and 10% update transactions. This workload will help me see how the caching mechanism performs in environments very similar to the ones in a large number of real world applications that exhibit read-dominated workloads [40];
- Workload B - a mixed workload with 50% read-only and 50% update transactions. This workload will help me evaluate the impact of an increase on the number of updates in the system.

Also, I configured the default broadcast rate of the BATCH strategy to 50 ms for all the benchmarks. Regarding the results, I compare the three invalidation strategies described in Section 4.3. with the normal execution of GMU, i.e., a execution without any caching mechanism. I consider three key performance indicators (KPIs) in the experimental study: throughput, cache hit percentange and total bandwidth consumed in the system.

5.4 TPCC Benchmark

5.4.1 Description

TPCC benchmark is a well-known On-Line Transaction Processing (OLTP) benchmark that portraits the activities of a whole-sale supplier that operates out of a number of warehouses and their associated sales districts. It is designed to scale just as the Company expands and new warehouses are created. The TPCC version used in this experimental study was adapted to run on top of transactional key-value stores (and used, in previous works to evaluate the performance of strongly consistent partial replication

[1] www.futuregrid.com

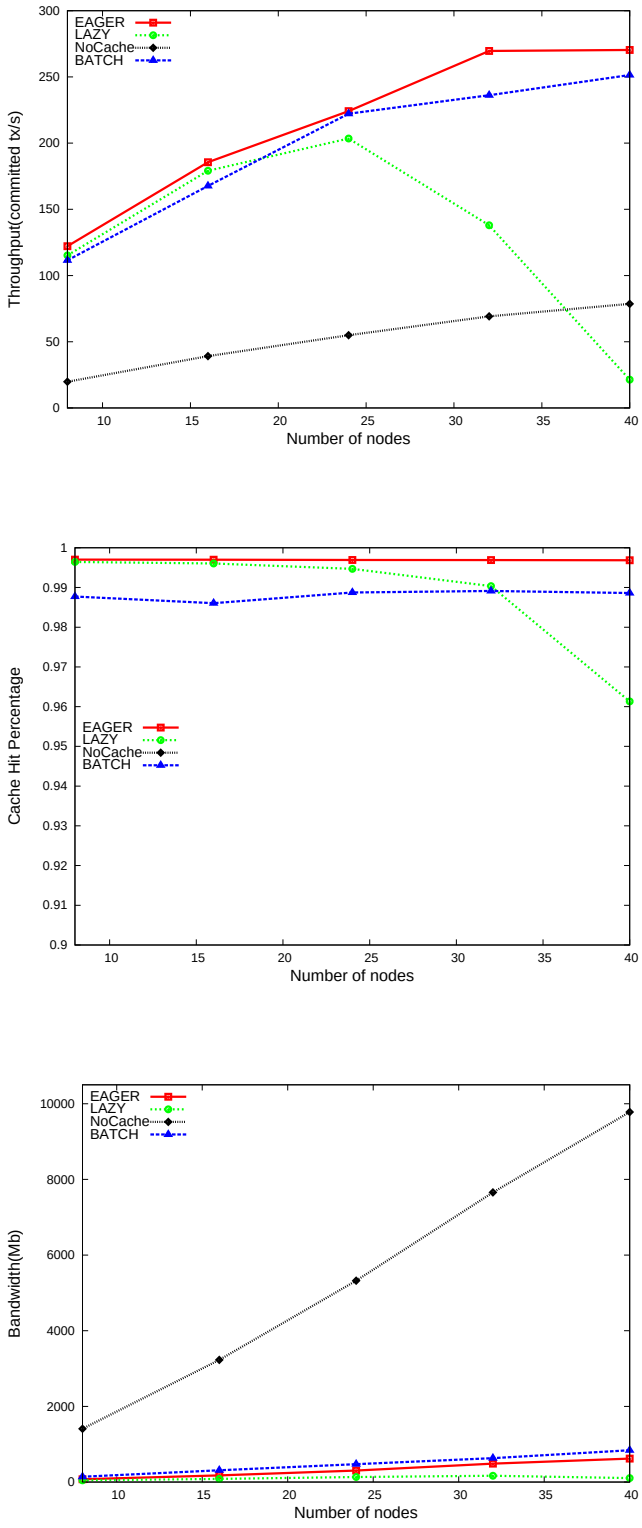

Figure 5.1: TPCC results obtained in FutureGrid for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

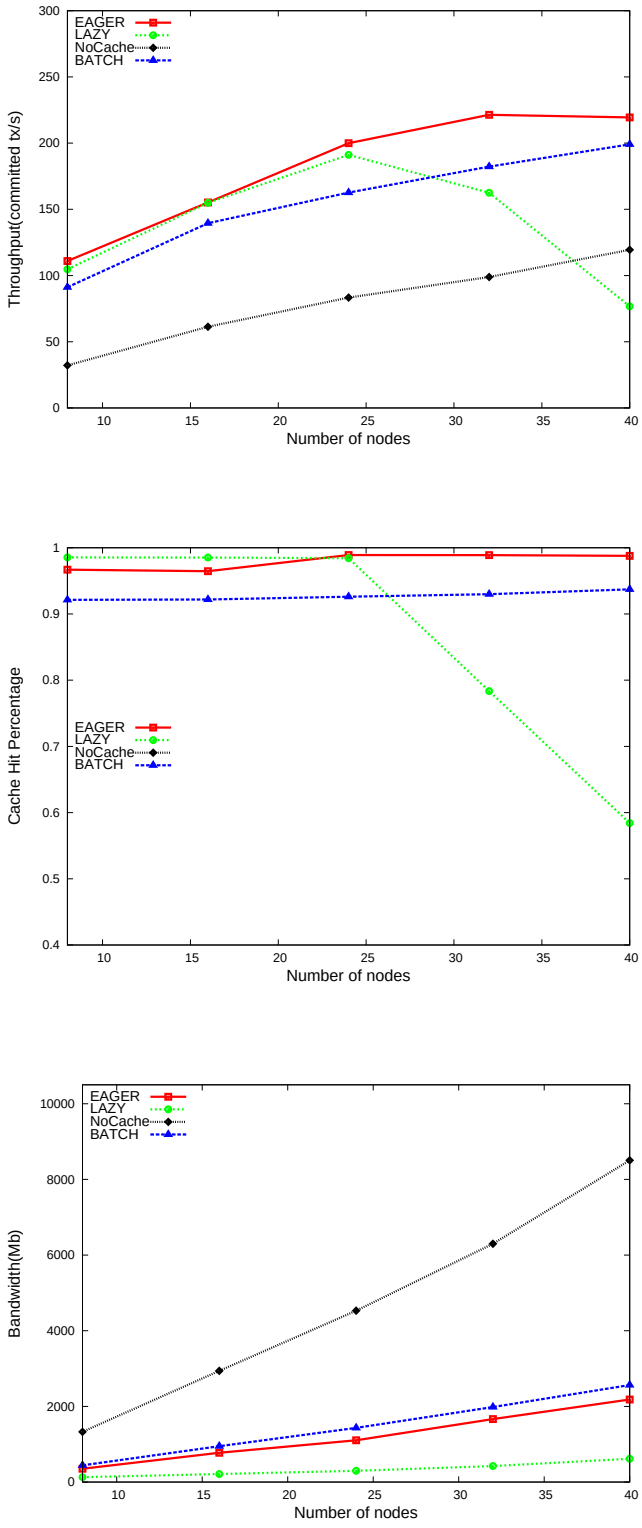

Figure 5.2: TPCC results obtained in FutureGrid for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

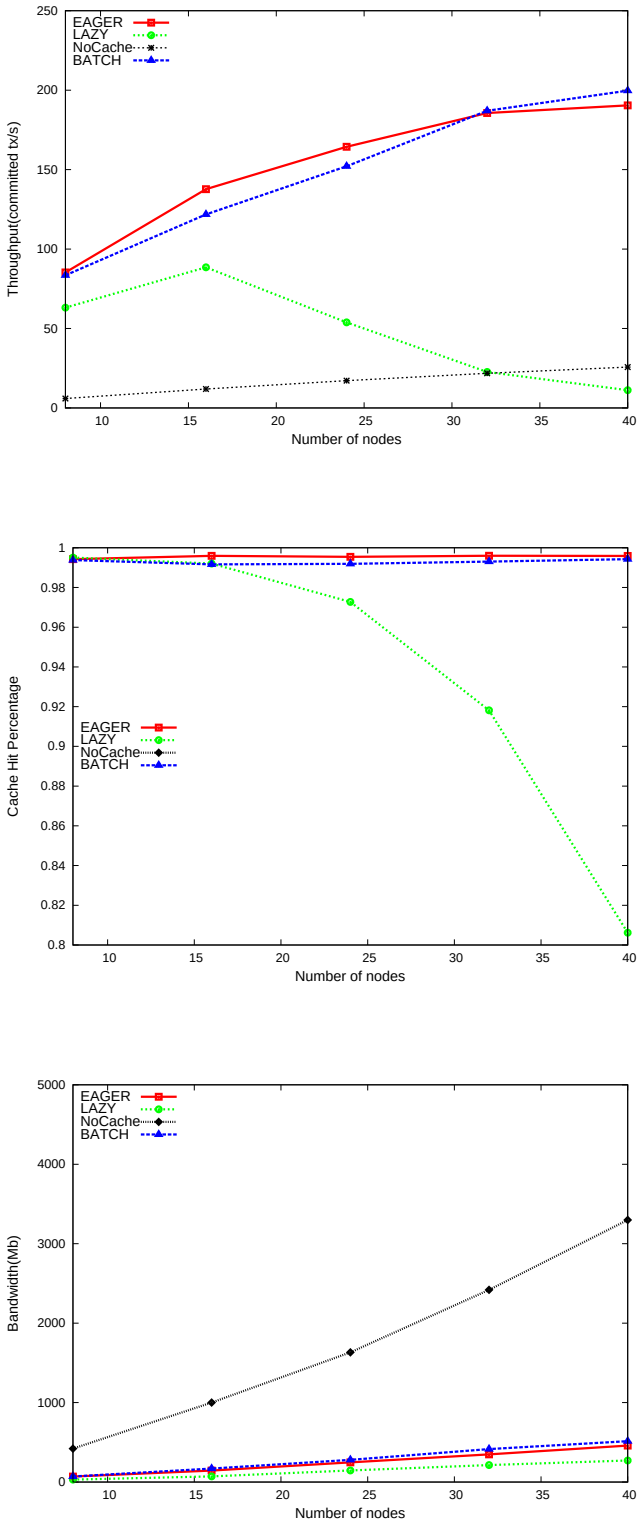

Figure 5.3: TPCC results obtained in CloudTM for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

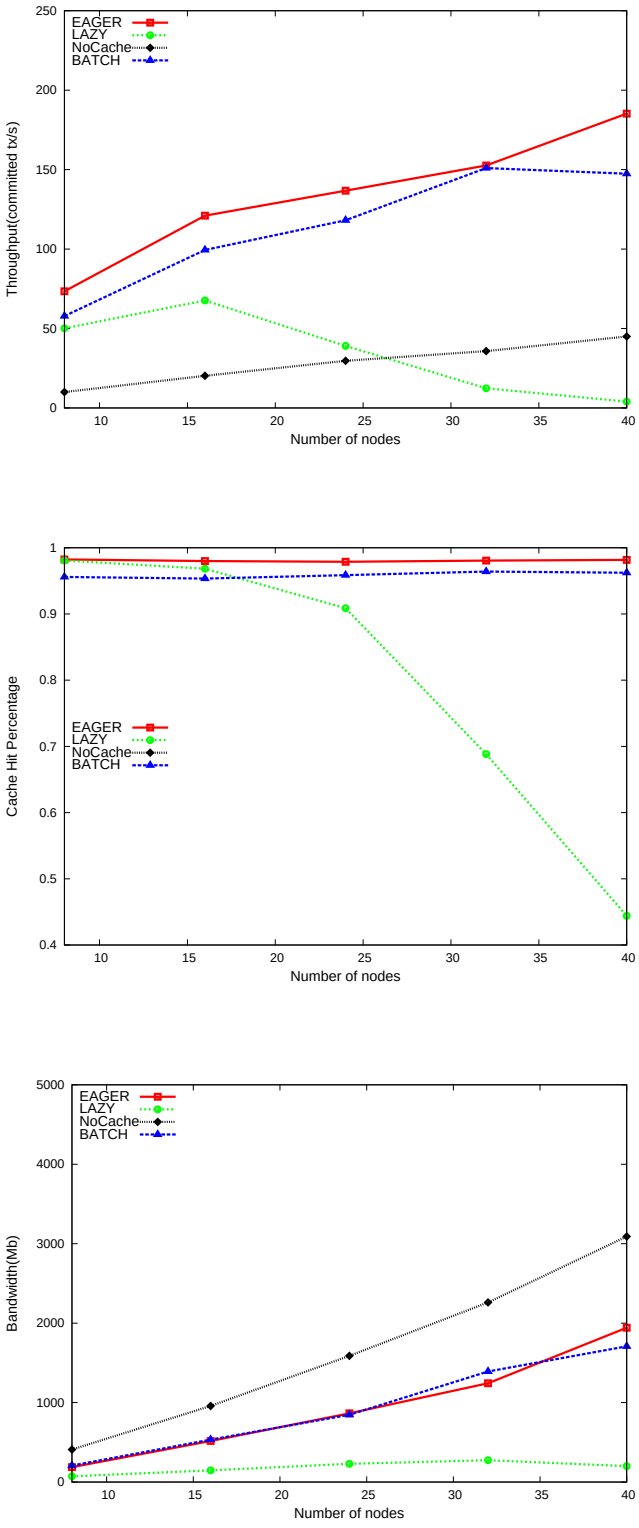

Figure 5.4: TPCC results obtained in CloudTM for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

protocols) and was configured to simulate an environment where each node corresponds to a warehouse and clients access the local warehouse with probability 75% and the neighbor warehouse with probability 25%. In other words, every transaction originated in a node accesses with probability 75% data local to that node and with 25% probability data remotely on the neighbor node. There are three distinct operations that can be done in this version of TPCC: OrderStatus, Payment and NewOrder. For Workload A I configured this three parameters with 90% OrderStatus, 5% Payment and 5% newOrder and for Workload B, 50%, 25% and 25%, respectively. Regarding the size of transactions, read-only transactions are very big so it is expected that the throughput will not be very high. Update transactions are relatively smaller than read-only transactions, however they are the biggest of all the benchmarks.

5.4.2 Results

Analyzing the results obtained in this benchmark, depicted in Figures 5.9., 5.10., 5.11. 5.12., it is clearly visible, in the throughput plots and in Table 1 (shows the best speedups and which strategy achieved them) in TPCC, that the EAGER strategy outperforms all the others. Looking at the cache hit percentage plots it is also clearly visible why, it has over 99% cache hits. BATCH is the second better strategy, also with over 99% cache hits but with slightly less throughput than EAGER. The LAZY strategy performs relatively well for the first set of nodes in all throughput plots however it does not scale. Looking at the cache hit percentage plot, it is also clearly visible why, the cache hits drops significantly when the size of the system grows. This happens because update transactions in TPCC invalidate a big number of keys and since LAZY is always waiting for a cache miss to apply the invalidation scheme, the probability of having cache miss is bigger when the number of keys invalidated is bigger. Also, the overhead introduced by the caching mechanism on the remote node can explain the drop on performance since the the construction and transmission of the invalidation messages lies on the critical path of transactions' execution. Regarding the total amount of bandwidth consumed by all the strategies, LAZY is where less bandwidth is consumed while EAGER and BATCHING have similar results. As expected NoCache consumes the most amount of bandwidth since it has to do remote requests whenever some data item is remote. Comparing the results on CloudTM and FG, as expected in CloudTM they are better in terms of speedup since caching provides higher throughput compared with no caching because the network is much more slower in CloudTM than in FutureGrid. However, even in FutureGrid the caching mechanism shows very decent speedups.

Regarding the abort rate, that was not presented in the plots because it was very low, it shows an increase of 2% in the LAZY strategy when using Workload B, which is expected for this strategy. The other strategies show similar abort rate (less than 4%).

	8	**16**	**24**	**32**	**40**
Workload A - CloudTM	14.3	11.5	9.5	8.5	7.8
Workload B - CloudTM	7.3	5.9	4.6	4.3	4.1
Workload A - FutureGrid	6.2	4.7	4.1	3.9	3.4
Workload B - FutureGrid	3.4	2.5	2.4	2.2	1.8

Table 5.1: Best speedups in TPCC. Red: EAGER, Blue: BATCH, Green: LAZY

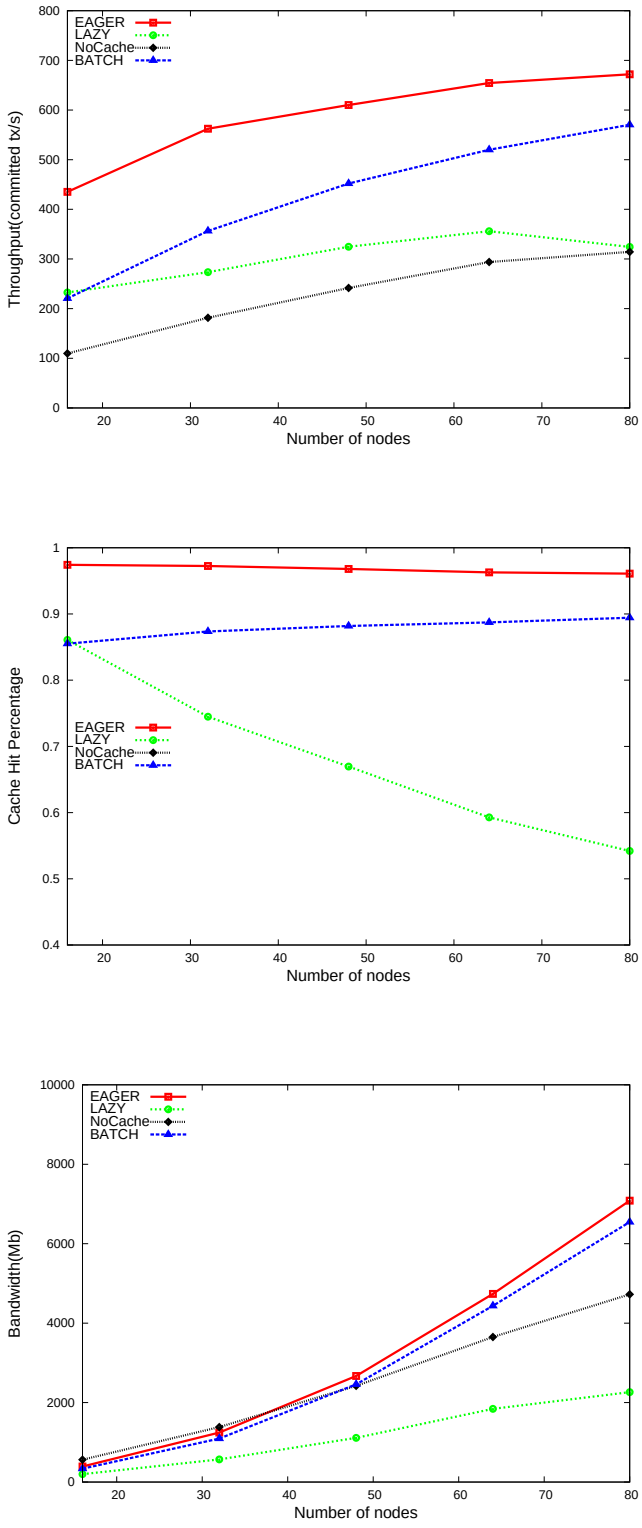

Figure 5.5: Vacation results obtained in CloudTM for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

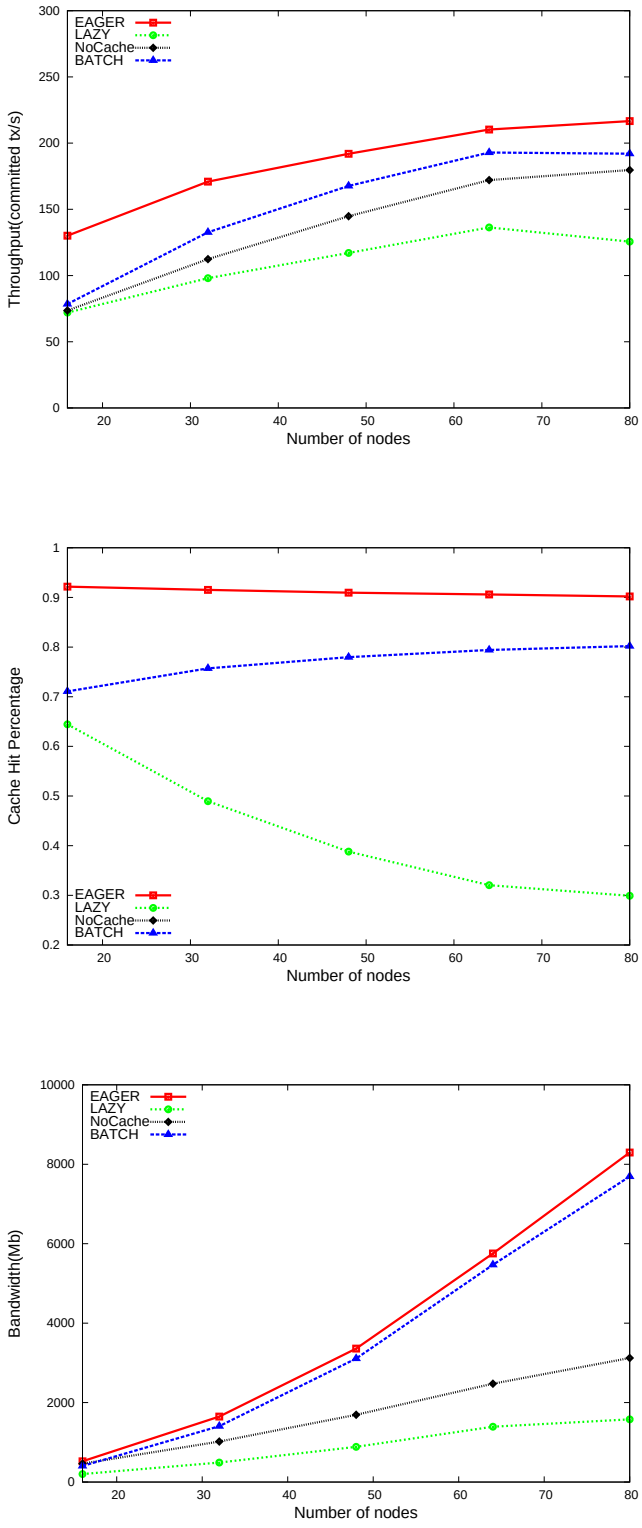

Figure 5.6: Vacation results obtained in CloudTM for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

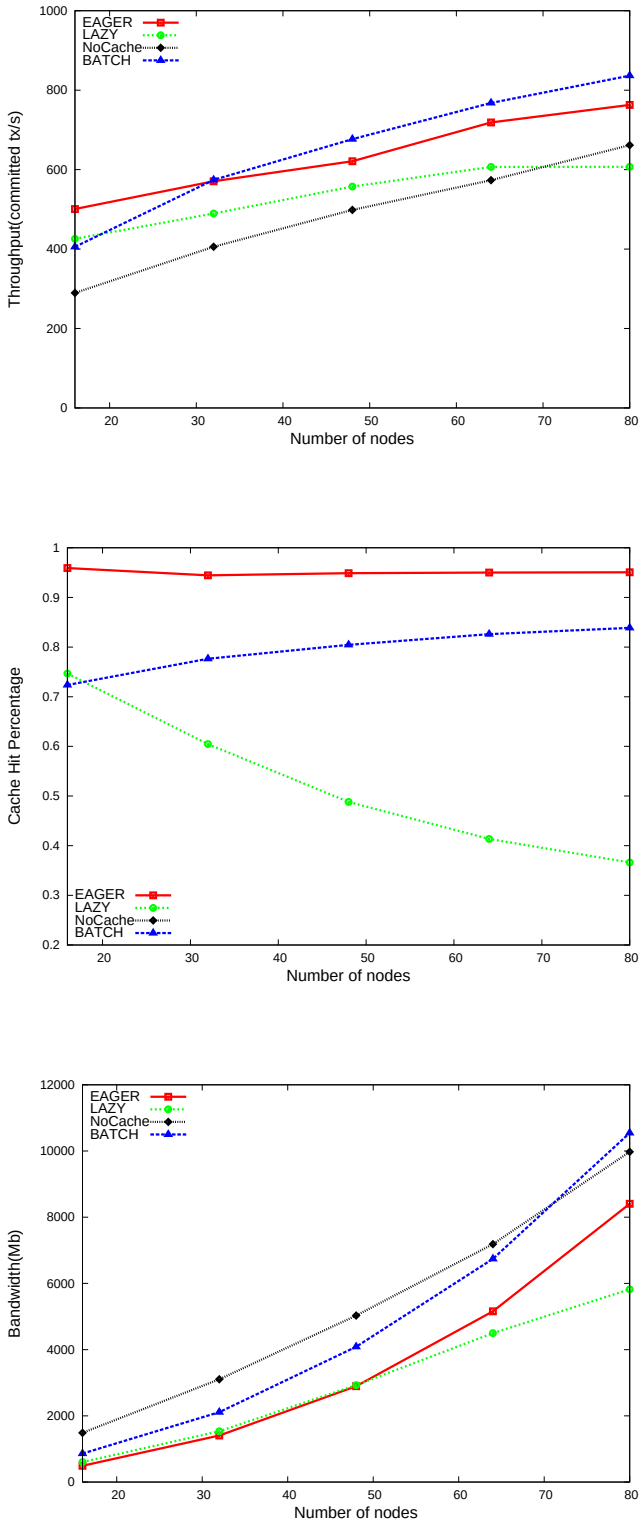

Figure 5.7: Vacation results obtained in FutureGrid for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

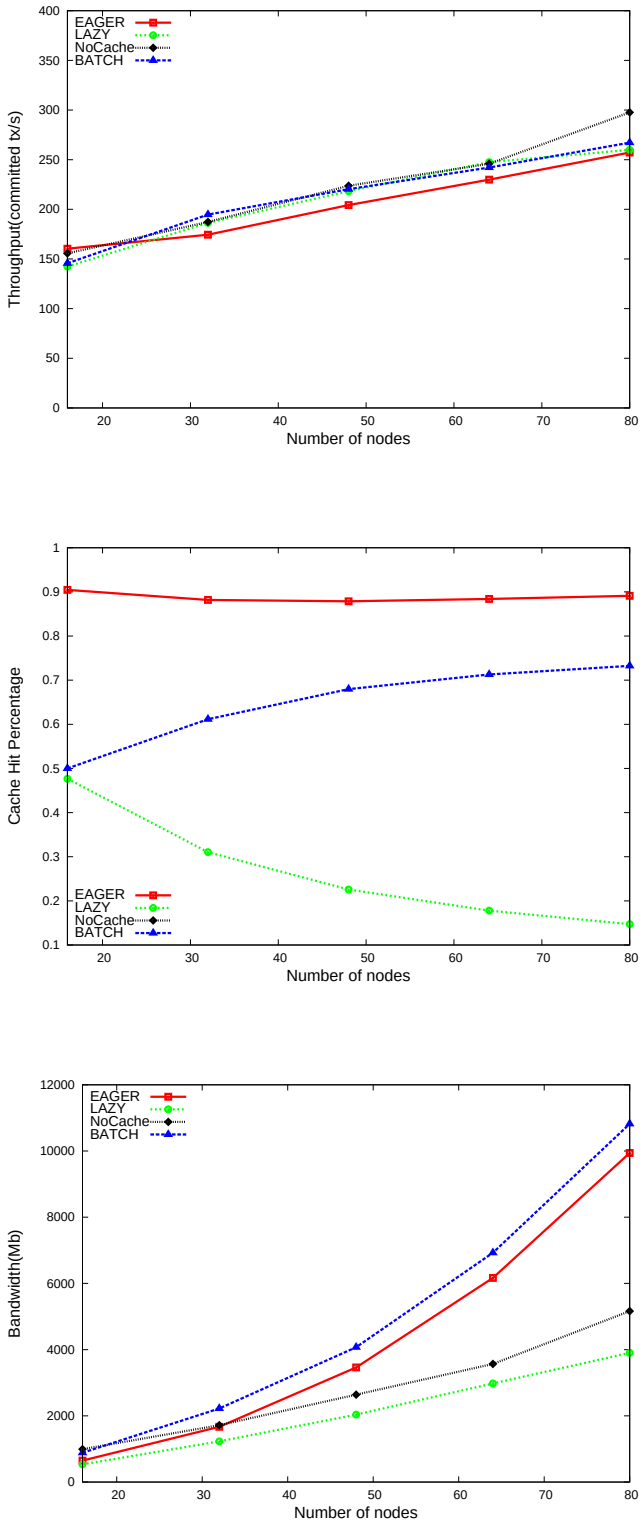

Figure 5.8: Vacation results obtained in FutureGrid for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

5.5 Vacation Benchmark

5.5.1 Description

Vacation is a well known benchmark from the STAMP [32]. It simulates an on-line travel agency in which several types of resources can be manipulated by customers or by the agency. There are three distinct types of sessions: reservations, cancellations, and updates. Like TPCC, I used a port of this benchmark for distributed key-value stores that is also designed to scale as the number of agencies grows. However, unlike the ported version of TPCC, the information of an agency is randomly placed in all nodes of the system using the default data placement scheme of Infinispan based on consistent hashing. The benchmark was configured to generate very little data contention, as the purpose of the experimental study is to analyze the effects of data locality on performance and scalability. Hence, by minimizing the likelihood of data contention, it is possible to evaluate more accurately the benefits achievable thanks to the usage of the caching mechanism proposed in this work. Furthermore, Workload A was configured using the parameter reservation, where a reservation can be an read-only transaction with probability 90% and an update transaction with probability 10%, while in Workload B the probabilities are 50% and 50%, respectively. The other parameters were configured to not be used. Regarding the size of transactions, both read-only and update transactions can be considered medium size comparing to TPCC.

5.5.2 Results

Analyzing the results obtained in this benchmark, depicted in Figures 5.13., 5.14., 5.15. 5.16., it is visible in the throughput plots and in Table 2, that the EAGER strategy outperforms all the others in CloudTM however, the margin is very small compared with the one in TPCC. Looking at the cache hit percentage plots is also visible why, it has over 96% cache hits. BATCH is again the second better strategy, also with over 96% cache hits but will slightly less throughput than EAGER. As in TPCC, the LAZY strategy performs relatively well for the first set of nodes in all throughput plots however it does not scale. Looking at the cache hit percentage plot, it is also clearly visible why, the cache hits drops significantly when the size of the system grows. In FutureGrid, BATCH is the better strategy but comparing it with NoCache it only has some speedup in Workload A. In Workload B, using either one of the strategies results in a similar or even worse performance than with NoCache. Looking at the amount bandwidth consumed by all the strategies, LAZY is again where less bandwidth is consumed, however EAGER, BATCHING and NoCache have very similar results. This is explained by the fact that transactions in this benchmark touch multiple nodes since data is place all around so every time a transaction commits on multiple nodes each one of them will issue an invalidation message. Also, the size of the keys can be an aspect to take in account since in Vacation they are 10 times bigger than in TPCC.

Regarding the abort rate, the results are negligible mainly due to how the benchmark was configured.

	16	32	48	64	80
Workload A - CloudTM	3.9	3.1	2.5	2.2	2.13
Workload B - CloudTM	1.8	1.4	1.3	1.3	1.3
Workload A - FutureGrid	1.7	1.4	1.4	1.3	1.3
Workload B - FutureGrid	1	1	1	1	0.9

Table 5.2: Best speedups in Vacation. Red: EAGER, Blue: BATCH, Green: LAZY

5.6 Synthetic Benchmark

5.6.1 Description

The synthetic benchmark was created by me to contain very small transactions (i.e., performs a very low number of read/write operations). Keys are randomly selected to fill up the transaction, so there is no actual business logic between two data access operations, which makes these transactions extremely short. I added the same notions of locality and neighbor node as in TPCC and configure it the same way, a transaction will access local data with probability 75% and remote data on the neighbor node with probability 25%. The amount of keys in the workload was configured to be 50000. Regarding the configuration of Workload A and B, it is used a parameter called readOnlyPercentage with the values 90% and 50%, respectively.

5.6.2 Results

Analyzing the results obtained in this benchmark, depicted in Figures 5.17., 5.18., 5.19. 5.20., it is visible, in the throughput plots and in Table 3 that that the LAZY strategy outperforms all the others in CloudTM and in Workload A of FutureGrid. Looking at the cache hit percentage plots it shows that LAZY and EAGER have very similar cache hits, however EAGER has much less throughput. This is clearly a result of the overhead introduced by the invalidation scheme of EAGER which can be seen in the total bandwidth consumption plots. Also, the BATCH strategy performs worse when compared to the other two benchmarks mainly because its cache hit percentage is not that high, which is clearly a sign that 50ms is not an ideal broadcast value in this benchmark. Regarding the abort rate, as in Vacation, the results are negligible due to the configuration of the benchmark.

	16	32	48	64	80
Workload A - CloudTM	2	2.8	3.1	3.8	3.8
Workload B - CloudTM	1.4	1.6	1.8	1.7	1.7
Workload A - FutureGrid	1.7	2.1	1.9	1.8	2
Workload B - FutureGrid	1	1.1	1.2	1.1	1.1

Table 5.3: Best speedups in Synthetic. Red: EAGER, Blue: BATCH, Green: LAZY

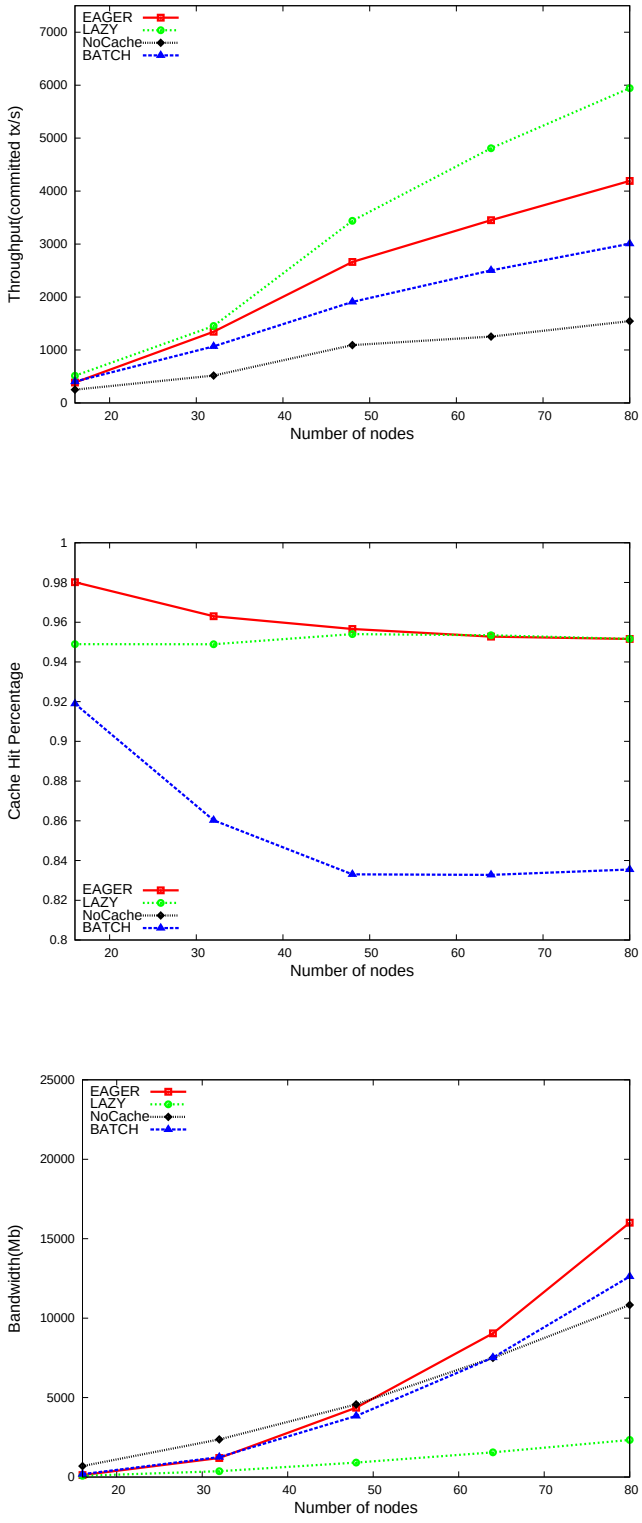

Figure 5.9: Synthetic results obtained in CloudTM for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

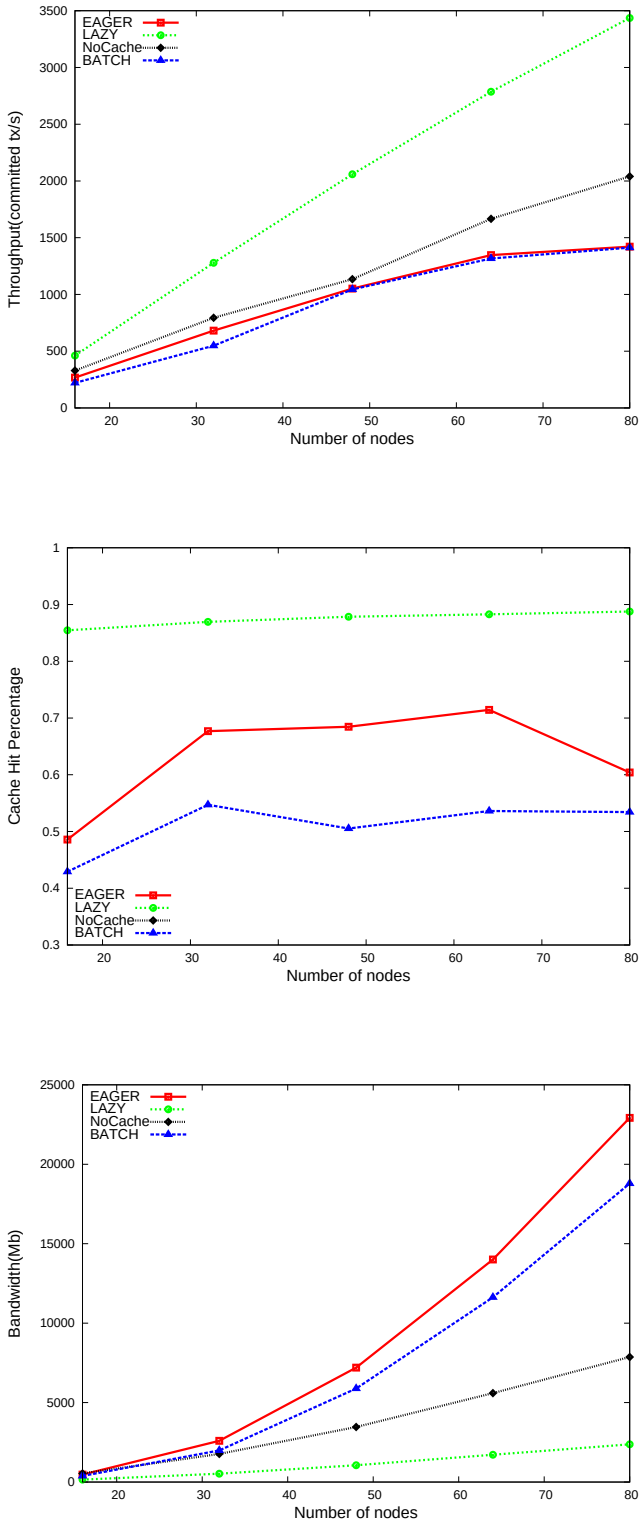

Figure 5.10: Synthetic results obtained in CloudTM for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

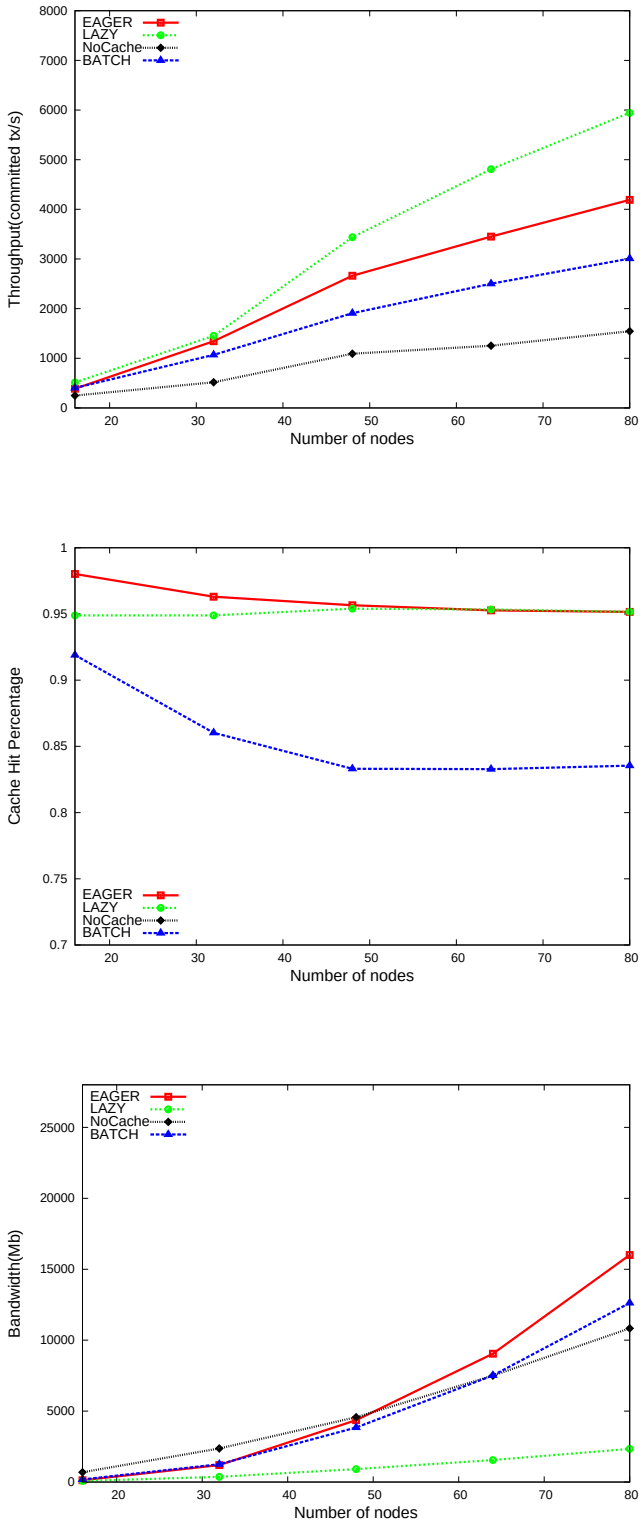

Figure 5.11: Synthetic results obtained in FutureGrid for Workload A. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

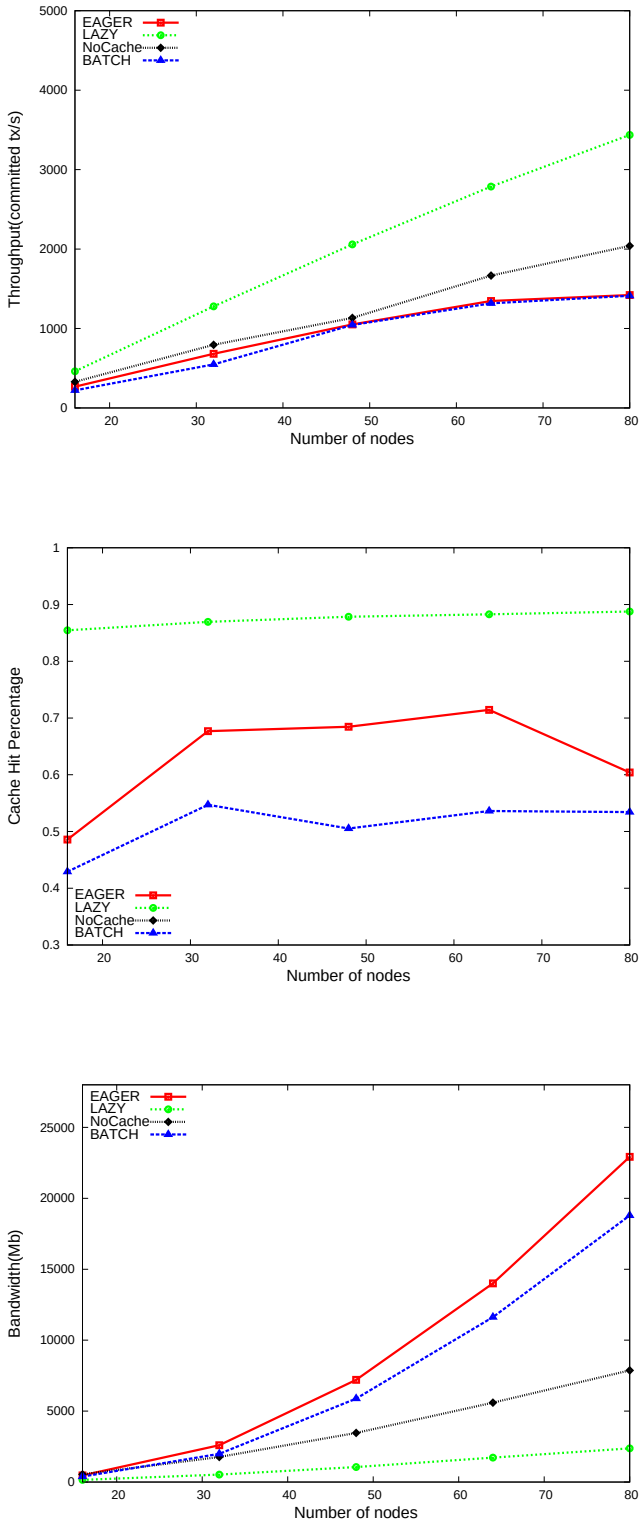

Figure 5.12: Synthetic results obtained in FutureGrid for Workload B. Top to Bottom: Throughput, Cache Hit Percentage, Total Bandwidth

.

5.7 Discussion

Summing up the evaluation chapter and answering to the questions in Section 5.1, it is clearly visible that GMU benefits a lot with the introduction of the caching mechanism. However, the various presented invalidation schemes used in the caching mechanism exhibit different trade-offs.

LAZY is attractive in network intensive workloads, where it allows for effectively saving bandwidth. However, by introducing in the critical path of transaction execution the construction and transmission of the invalidation messages, it can incur in large overheads in workloads that generate large iSet messages.

EAGER is more effective in ensuring high hit rate, especially for applications where read-only transactions are long and the number of keys in update transactions is big, but for smaller transactions the overhead introduced can be detrimental even when having a very good cache hit rate.

BATCH, despite having the potential for reducing communication and, hence, enhance efficiency, did not prove to be particularly beneficial in the considered workloads. The gains in terms of reduced bandwidth with respect to eager, in fact, are normally outweighed by the drop in the cache hit rate imputable to the delays induced by batching the cache invalidation messages.

The abort rate had relatively no impact on the results presented, however this happened mainly due to how the environments and workloads were configured.

Chapter 6

Conclusions

The potential for scalability of partial replication protocols that use consistent hashing data placement can be severely hampered when the applications' data access patterns do not exhibit a good degree of locality. GMU is one of those cases.

In this dissertation, I introduced in GMU a caching mechanism that relies on an invalidation scheme that can use multiple dissemination strategies to maintain the data fresh, to help enhance the locality of data.

I conducted an extensive experimental study to analyze the efficiency and effectiveness of the proposed caching mechanism. From the experimental results it is visible that the different strategies used have different performances depending on the environment and the workload, motivating further research in the area of adaptive caching schemes.

Concluding, it is clearly visible that with the introduction of the caching mechanism, data locality was enhanced thus the overall performance of the system was also improved.

6.1 Future Work

There is so much that can be done to improve this work. The strategies described in Section 5.3 can be used as basis for exploring different trade-offs in the design of cache invalidation schemes, which are not as eager as EAGER an not as lazy as LAZY. Specifically, EAGER disseminates everything that is committed even to nodes where that information is not needed, so the new strategy should know what is the information that each node has and disseminate only that information. On the other hand, LAZY always waits for a cache miss to leverage the invalidation scheme. For instance, a new strategy could be a mix of EAGER and LAZY that disseminates information only to nodes where transactions that touch multiple nodes commit, at commit time, to diminish the probability of having a cache miss in the next transaction originated on those nodes. Regarding BATCH, since the results did not show particularly beneficial in the considered workloads, a more comprehensive study could be carried out. In this sense, it would be interesting to assess to what extent performance could increase by adopting adaptive strategies for self-tuning the batching dissemination rate. This problem has indeed resemblances with

the problem of tuning the batching latency in total-order broadcast protocols, and has been approached using techniques from control theory [17, 3] and machine learning [41]. It is expected that such schemes may be particularly beneficial in presence of dynamic workloads.

Another interesting aspect is to explore a similar idea to the one described in Section 4.2.3. but with the purpose of minimizing the chances of incurring in a cache miss by allowing to initialize the transaction vector clock of read-only transactions with a conservative value obtained as the minimum of all the validity values gathered by the invalidation scheme. The vector clock is also ”blocked”, i.e., it is marked as if the transaction had already ready from all nodes in the system, which guarantees that it will not be advanced during the transaction’s execution. Setting the transaction’s value in this way corresponds to serialize it before the oldest snapshot present in the local cache node. This can enhance the likelihood that the transaction can access ”fresh enough” data in the cache.

In general, the use of an adaptive caching scheme using all the strategies described above and the ones already implemented, would be an interesting research subject.

Bibliography

[1] ADYA, A. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Tech. rep., PhD Thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 1999.

[2] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[3] BARTOLI, A., CALABRESE, C., PRICA, M., ANTONIUTTI, E., MURO, D., AND MONTRESOR, A. Adaptive message packing for group communication systems. In *In On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops, volume 2889/2003 of Lecture Notes in Computer Science* (2003), Springer-Verlag GmbH, pp. 912–925.

[4] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data* (1995), SIGMOD '95, ACM, pp. 1–10.

[5] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.

[6] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., AND TOUEG, S. Distributed systems (2nd ed.). ACM Press/Addison-Wesley Publishing Co., 1993, ch. The primary-backup approach, pp. 199–216.

[7] CACHIN, C., GUERRAOUI, R., AND RODRIGUES, L. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[8] CACHOPO, J., AND RITO-SILVA, A. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63* (December 2006), 172–185.

[9] CARVALHO, N., ROMANO, P., AND RODRIGUES, L. Scert: Speculative certification in replicated software transactional memories. In *The 4th Annual International Systems and Storage Conference* (2011), IBM Research.

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In

Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (2006), OSDI '06, USENIX Association, pp. 15–15.

[11] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*, 4 (Dec. 2001), 427–469.

[12] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 251–264.

[13] COUCEIRO, M., ROMANO, P., CARVALHO, N., AND RODRIGUES, L. D^{2}STM: Dependable distributed software transactional memory. In *Proc. of PRDC* (2009), IEEE CS, pp. 307–313.

[14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (2007), SOSP '07, ACM, pp. 205–220.

[15] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*, 4 (Dec. 2004), 372–421.

[16] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing* (2006), DISC'06, Springer-Verlag, pp. 194–208.

[17] DIDONA, D., CARNEVALE, D., GALEANI, S., AND ROMANO, P. An extremum seeking algorithm for message batching in total order protocols. In *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on* (2012), pp. 89–98.

[18] ELNIKETY, S., DROPSHO, S., AND ZWAENEPOEL, W. Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 399–412.

[19] FRANKLIN, M. J. *Caching and memory management in client-server database systems*. PhD thesis, 1993.

[20] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (1996), SIGMOD '96, ACM.

[21] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proc. of PPOPP* (2008).

[22] HADZILACOS, V., AND TOUEG, S. Distributed systems (2nd ed.). ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant broadcasts and related problems, pp. 97–145.

[23] HAERDER, T., AND REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (Dec. 1983), 287–317.

[24] HANSDAH, R. C., AND PATNAIK, L. M. Update serializability in locking. In *ICDT 86, International Conference on Database Theory, Rome, Italy, September 8-10, 1986, Proceedings* (1986), vol. 243 of *Lecture Notes in Computer Science*, Springer, pp. 171–185.

[25] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), OOPSLA '06, ACM, pp. 253–262.

[26] J. PAIVA, P. RUIVO, P. R., AND RODRIGUES, L. Autoplacer: scalable self-tuning data placement in distributed key-value stores. In *In Proceedings of the 10th International Conference on Autonomic Computing (ICAC '13)* (San Jose, CA, USA, June 2013).

[27] JIANG, S., AND ZHANG, X. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2002), SIGMETRICS '02, ACM, pp. 31–42.

[28] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[29] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[30] LIM, J., CHUNG, J., KIM, J., AND SHIM, K. A dynamic load balancing for massive multiplayer online game server. In *ICEC* (2006), R. H. R. Harper, M. Rauterberg, and M. Combetto, Eds., vol. 4161 of *Lecture Notes in Computer Science*, Springer, pp. 239–249.

[31] MARCHIONI, F., AND SURTANI, M. *Infinispan Data Grid Platform*. Packt Publishing.

[32] MINH, C. C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. Stamp: Stanford transactional applications for multi-processing. In *IISWC* (2008), IEEE, pp. 35–46.

[33] OOI, B. C. Cloud data management systems: Opportunities and challenges. *Semantics, Knowledge and Grid, International Conference on 0* (2009).

[34] PALMIERI, R., QUAGLIA, F., AND ROMANO, P. AGGRO: Boosting stm replication via aggressively optimistic transaction processing. *Proc. of NCA* (2010), 20–27.

[35] PEDONE, F., GUERRAOUI, R., AND SCHIPER, A. The database state machine approach. *Distributed and Parallel Databases 14*, 1 (July 2003), 71–98.

[36] PEDONE, F., AND SCHIPER, A. Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci. 291*, 1 (Jan. 2003), 79–101.

[37] PELUSO, S., ROMANO, P., AND QUAGLIA, F. Score: a scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference* (New York, NY, USA, 2012), Middleware '12, Springer-Verlag New York, Inc., pp. 456–475.

[38] PELUSO, S., RUIVO, P., ROMANO, P., QUAGLIA, F., AND RODRIGUES, L. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS* (2012), IEEE, pp. 455–465.

[39] POWELL, D. Group communication. *Commun. ACM 39*, 4 (Apr. 1996), 50–53.

[40] ROMANO, P., CARVALHO, N., AND RODRIGUES, L. Towards distributed software transactional memory systems. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware* (New York, NY, USA, 2008), LADIS '08, ACM, pp. 4:1–4:4.

[41] ROMANO, P., AND LEONETTI, M. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In *Computing, Networking and Communications (ICNC), 2012 International Conference on* (2012), pp. 786–792.

[42] SCHIPER, N., SCHMIDT, R., AND PEDONE, F. Optimistic algorithms for partial database replication. In *Proceedings of the 10th international conference on Principles of Distributed Systems* (2006), OPODIS'06, Springer-Verlag, pp. 81–93.

[43] SCHIPER, N., SUTRA, P., AND PEDONE, F. P-store: Genuine partial replication in wide area networks. In *Proc of SRDS* (2010), IEEE CS, pp. 214–224.

[44] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[45] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (1995), PODC '95, ACM, pp. 204–213.

[46] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), SOSP '11, ACM, pp. 385–400.