Serializable and Rollback-Resilient Transaction Protocols using Trusted Counters

Dimitra Giantsidi

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2019

Abstract

Hardening the security semantics of software systems running on the untrusted computing infrastructure (such as cloud, IoT and edge computing resources), has always been a pole of attraction for the research community. The recently wide development of hardware-assisted *Trusted Execution Environments* (*TEE*) (such as Intel SGX [6] and ARM Trustzone [1]) opens new horizons towards the design of secure software systems.

This thesis addresses security violations known as *rollback* or *replay* attacks [35] whose main target is to revert a system to a stale, obsolete state. This can be achieved by arbitrarily shutting down the system and tampering with the log files or other persistent state-related data. Particularly, our work aims to design secure transaction protocols that safeguard transaction processing against rollback attacks while ensuring transactions' *serializability* [31]. The latter property is necessary when concurrent transactions are processed since it ensures isolation and hence system's correctness.

We developed our algorithms in RocksDB, a Key-Value store [12]. For serializability and rollback-protection we take advantage of the state-of-art hardware counters provided by TEE such as SGX. However, in this work we identify the limitations of such counters which make their use impractical for high-throughput systems, like ours, that prevail in the cloud. We overcome these limitations by leveraging asynchronous trusted counters that have been proposed by SPEICHER [36]. Judging from the obtained results, the use of the asynchronous trusted counters is irreplaceable in order to achieve high throughput. Additionally, our evaluation shows that our proposed system scales good with the number of clients. The performance overhead compared to a system without rollback protection can only be considerable with minimum parallelism. Indeed, under workloads with long transactions and moderate to high number of concurrent clients, our system presents little or no overhead compared to a system without security guarantees.

Acknowledgements

This thesis signifies the completion of my MSc studies, an important step in both my academic career and my life in general.

I would like to express my gratitude towards my advisor Pramod Bhatotia first, who offered me the opportunity to join his research group and work on this very interesting problem. Without his guidance through this project, this thesis would not have the current form. I would also like to thank him for the excellent lectures he delivered on the "Extreme Computing" course, for sharing his knowledge with us and inspiring me to want to contribute more in this field. I would also like to thank Maurice Bailleu for his help, time, patience and willingness to discuss all my questions, even the most trivial ones. Lastly, I would really like to thank Natacha Crooks for her time to proofread this thesis and provide me with feedback. Her comments were extremely useful.

I want to thank all my friends that supported me in my decision to study abroad and especially Foivos for always encouraging me to achieve the best of myself. Last but not least, I thank my family for providing me the supplies to pursue this degree and for their unconditional support and love.

I wish this work to be the starting point for many future good things.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

With the high adoption of the cloud infrastructure, the security violations have been increased [41]. Services and data remain susceptible to adversaries that may exploit software bugs of privileged code [24]. To protect data processing, a secure system must safeguard both the running software and the state of the data this specific software operates on. In this work, we address a specific type of violations that aim to revert the system to a stale state. For instance, an adversary, by arbitrarily shutting down the system and compromising log files, may successfully rollback the system to a correct but obsolete state. Such attacks are defined as *rollback* or *replay* attacks [35].

Rollback attacks have severe implications when data *freshness* is necessary. A motivating example could be the case of a financial service which continuously receives transactions that update stored data, e.g. account balances. A successful rollback attack is possible to revert the system to an obsolete state that does not match the executed transactions. Consequently, the history of banking transactions and the account balances may be not correct. Auctions, voting, control systems, databases and storage systems also showcase the importance of data freshness.

Towards the direction to improve the security properties of applications, Intel has launched *Software Guard Extensions* (*SGX)* [6] to its latest processors. SGX-enabled processors can isolate selected code and data from privileged code ensuring data *integrity* and *confidentiality* while they also provide trusted hardware counters (*SGX-counters*) that are extremely popular to ensure rollback protection.

Additionally, modern database systems [12, 8, 3, 5, 2] allow operations to overlap in time for higher throughput. The major correctness criterion for parallel oper-

ations, is *serializability* [31] as it provides isolation between these concurrent operations, ensuring that the state of the system is always valid. In transaction processing (e.g. databases, transactional memory [25], software transactional memory [43]) both centralized and distributed, a *transaction schedule* is *serializable* if its result (e.g. its outcome on the database) is equivalent to the result of executing its transactions serially in time without any overlapping among them [31, 32].

This work addresses both serializability and rollback protection challenges in transaction processing in storage systems. More presicely, we leverage the well-known counters-based technique for rollback resilience in combination with the SGX-counters to also serialize the commit order of all issuing transactions. In our project, we investigate the use of native SGX-counters addressing their potential limitations that encourage us to use asynchronous trusted counters similar to those proposed by SPEICHER [36]. Our algorithms have been developed on top of RocksDB [12], a Key-Value store.

1.2 Thesis Contribution

The contributions of this work are:

1. the design of serializable and rollback-resilient transaction protocols under Pessimistic and Optimistic Concurrency Controls. Highly inspired by distributed transactions protocols [19, 21] with strong serializability semantics, we implement our algorithms into a single node database. However, assuming only Pessimistic and Optimistic Concurrency, makes our principles applicable to a distributed environment as well.

2. the design of a crash recovery algorithm that detects rollback attacks' violations and respects the committed transactions' order.

3. the performance evaluation and analysis of our protocols compared to *a.* a system without rollback-security guarantees and *b.* a system which achieves these properties using SGX hardware counters.

1.3 Chapter Outline

Chapter 2 presents the theoretical background which is considered important for a reader of this work in order to comprehend the concepts and designs discussed later.

Chapters 3 and 4 describe our design and implementation details, respectively. We give a brief overview of our proposed system. We discuss the limitations of naively using the SGX-counters to achieve serialization and rollback-protection. Following this, we present our protocols that overcome these issues leveraging asynchronous counters. Our protocols achieve serialization, rollback-protection and crash resilience. Lastly, we present the recovery algorithm which is adapted to detect rollback attacks.

In Chapter 5, we discuss the evaluation methodology. Precisely, we present a basic analysis on a native system's instance and thereafter we compare it with our proposed system with rollback protection. We analyse the results of our experiments and draw conclusions regarding our implementation such as the performance impact of the SGX-counter and the asynchronous counter.

Chapter 6, summarizes and critically evaluates our work. We also discuss future directions of this project.

Appendices present additional information we collected during our time working on this project. Even though this knowledge is not really necessary for a reader to comprehend the current project's idea, we believe that it arises interesting questions and future directions for our work.

Chapter 2

Background

2.1 Rollback Attacks and Intel SGX Technology

Rollback attacks are security violations in which malicious entities compromise the state of the system by restarting it and replaying an older log. The recovered system's state, even stale, might be consistent, thereby making rollback attacks hard to be detected.

To address this problem, *monotonic trusted hardware counters* have been introduced [35]. In brief, once the counter's value advances, it cannot be reverted back. Hence, the latest version of the system can always be shown by the counter's value.

Two are the common techniques for counter-based rollback protection: *inc-then-store* and *store-then-inc* [35]. The first approach denotes that the h/w counter is incremented and then, its value is stored along with the data in the disk. This technique provides the strongest security semantics but the recovery after a crash is hard. In case the system crashes after the counter's increment but before the store procedure, at recovery the h/w counter will point to a future, not yet been saved in the disk, value. Consequently, it cannot be proved whether the state in the disk is the most recent.

In the second approach, the system's state is first saved on the disk along with an input value and then the counter is updated to that input. Once the counter's update completes, the state in the disk is rollback protected. In contrast to the first approach, the later offers crash resilience; the h/w counter will never exceed the latest stored state in the disk.

In the long line of trusted computing [15, 23], *Intel Software Guard Extensions* (*SGX*) [6] – x86 ISA extensions for *Trusted Execution Environment* (*TEE*)– seem to be off-the-self technology. SGX-enabled architectures offer a trusted hardware which

establishes a secure container. Selected code and sensitive data are loaded to this secure container. This container is called *enclave* or *secure memory enclave* and it is a specific, hardware-protected memory region (at most 128 MB) that isolates the execution from privileged code guarantying confidentiality and integrity – even on compromised platforms – as long as the computation runs in it. A large body of research has been concentrated on taking advantage of SGX [29, 35, 18, 37, 38, 36, 39, 17, 42]. More details about SGX technology can be found in Appendix B.1.

Recently Intel has also embed to its platforms h/w monotonic counters [7], providing the engineers with a building block to safeguard their applications against rollback attacks. However, researchers [38, 35, 36] have raised concerns about SGX-counters' performance and durability. As we discuss in Section 3.2, writes to SGX-counters are slow (60-250ms) while their continuous use may lead them to an unrecoverable state.

2.2 Threat Model

Similarly to the SGX standard threat model [18], we consider a powerful adversary who is able to get the control of the entire software stack including privileged code (OS, hypervisor). In our work, we do not provide security guarantees for data integrity and confidentiality. Our threat model considers only rollback attacks, for example, adversaries that arbitrarily shut down the system and tempting to recover from a stale state (by replaying an old log). Additionally to this, we also detect adversaries that might tamper with log files such as deleting, duplicating and/or reordering log entries.

2.3 Literature Review on Distributed Transactions and Secure Storage Systems

Our design is highly inspired by Spanner [19], a geo-distributed replicated database, without security guarantees, that uses a wall-clock time to achieve global serializability and external consistency [31]. Spanner, introduces the *commit-wait* operation which delays the commit phase of ongoing operations until an *uncertainty period* passes. That way, Spanner efficiently deals with clock drifts that are present in a distributed infrastructure while ensuring strict consistency. The commit-wait takes place while locks are held, so no dependent operations can proceed before the ongoing operation completes. The performance degradation is proportional to the uncertainty period.

Another important work is Obladi [21], a KV store that serializes distributed transactions and, additionally, uses ORAM [22] to hide access patterns. Obladi splits time into *epochs*, fixed size periods of time, which are used to optimize read and write paths. However, both start and commit phases of a transaction should be in the same epoch. Lastly, they add protection against rollback attacks by assuming the existence of a persistent trusted counter without however implementing it.

Priebe et al. [39] propose EnclaveDB, a secure in-memory SQL-database that leverages SGX to protect itself against a powerful adversary [18]. EnclaveDB, however, does not consider enclaves' limited size; they assume that enclaves are able to host both the entire engine and sensitive data (public data are stored in the untrusted area). They address rollback attacks with h/w monotonic counters and they serialize transactions by assigning monotonic counters' values to both start and prepare phases.

SPEICHER [36] and ShieldStore [29], are both KV stores that expose data integrity and confidentiality semantics with SGX. SPEICHER also implements an asynchronous counter (Section 2.5) for rollback attacks' resilience. Their main difference is that ShieldStore is an in-memory database while SPEICHER extends security in the persistent storage. Neither of the systems supports transactions but they both support query operations and tackle with the limited enclave size. Their approaches reassemble; any in-memory data is kept encrypted in the untrusted memory. Particularly, SPEICHER places keys and the hash of the value inside the enclave while values are encrypted in the untrusted area. In contrast, ShieldStore keeps only critical meta-data in the enclave and all KV pairs are encrypted in the untrusted memory.

2.4 RocksDB

2.4.1 System Overview

In our project, we use RocksDB [12], a persistent Key-Value (KV) store that is optimized for fast storage devices such as flash drives and high-speed disk drives. An overview of the RocksDB System is shown in Figure 2.1. RocksDB stores data persistently in SSTable files [16]. These files are immutable, organized into levels with increasing sizes, and sorted by keys. To quickly serve requests and speed up any writes to the database, RocksDB keeps recent updates in an in-memory data structure, called MemTable. The default MemTable implementation is a Skip List [40] that offers fast and concurrent accesses. Once the MemTable grows up to a configurable limit, it is

stored as an SSTable file in the persistent storage. At this point, RocksDB may perform *compaction* to re-organize keys in a sorted fashion in the SSTable hierarchy. Sorting keys encourages the binary search withing each SSTable upon a data request.

RocksDB offers durability by logging in advance the updates to a Write-Ahead-Log (WAL) in the persistent storage; every update in the database is first written to the WAL and then to the MemTable. In case the system crashes before the MemTable has been completely updated, we can rebuild the latter by replaying the WAL. Lastly, RocksDB makes use of a second file called Manifest which logs state updates, such as compactions, the creation and the deletion of SSTable files or WAL files. Upon restart after a crash, log files are replayed to establish the system's latest state.

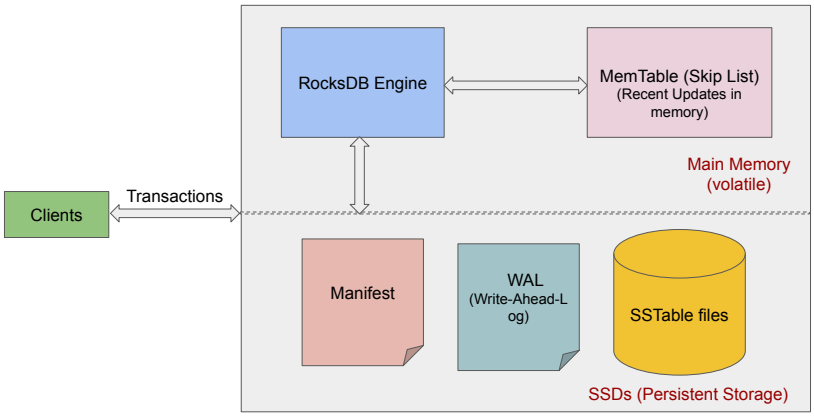

Figure 2.1: RocksDB System Overview.

Information about RocksDB's *logging protocol* can be found in Appendix A.2.

2.4.2 Transactions in RocksBD

RocksDB supports both query operations and transactions. As far as transactions are concerned, RocksDB is adaptable to different workloads by supporting two concurrency controls; *Pessimistic* and *Optimistic Concurrency Control*.

Under Pessimistic Concurrency Control (PCC), RocksDB acquires incrementally the locks of all keys the current transaction is going to write to the database. Such a locking mechanism is a typical approach to deal with conflicts and achieve isolation. In case a transaction cannot lock a key – that is normally happening when another ongoing transaction is currently holding the lock – the current transaction will continuously try to lock it for a specific *timeout period*. If eventually the lock cannot be acquired, the

transaction aborts (timeouts). This timeout period successfully resolves deadlocking situations, however, a client can optionally, enable the deadlock detection mechanism (disabled by default) which will handle deadlocks by traversing the dependency graph in the *Key Lock Manager*.

PCC is recommended for workloads with concurrency and conflicts. However, the locking mechanism introduces overheads since the PCC will check for conflicts for all write operations, including writes performed outside of a transaction. In default PCC configurations, RocksDB implements a *WriteCommitted* write policy, according to which transactions are written to the database and become visible to other clients only after they successfully commit. This policy is simple and intuitive but may be problematic for long transactions or multiple parallel clients because the memory can be a bottleneck (all transactions buffer their updates until they commit). This observation, in combination with the fact that RocksDB does not support parallel logging (Appendix A.2), can lead to lengthy commits and decrease system's throughput.

To tackle both limitations, two other policies have been designed, *WritePrepared* and *WriteUnprepared*, which attempt to write the updates into the database at prior stages. The challenging parts in these policies are to ensure consistency and correctness, especially for the read path; all clients must *see* the same state of the system and the data should not be visible until the transaction successfully commits (Appendix A.1). In our project, we implement our protocols under default settings.

Optimistic Concurrency Control (OCC) provides a more light-weight conflict detection mechanism. OCC does not take any locks, allowing conflicting transactions to be issued in parallel. The conflict detection takes place at transactions' commit-phase, while at the meantime, all updates are buffered locally. The conflict detection validates that no other writers have modified an ongoing transaction's keys in the meantime. If there is a conflict with another write or it cannot be determined[1], the transaction aborts. Due to its nature, OCC is only recommended for workloads with very sparse write conflicts and little or small transactions since they, similar to pessimistic transactions with the default write policy, can still saturate the memory. RocksDB currently does not support alternative write policies for OCC.

[1] A client can configure whether the conflict detection will also check for conflicts in the SSTable files. Normally, only the recent updates present in the MemTable are checked for higher performance.

2.5 SPEICHER's Asynchronous Monotonic Counter

SPEICHER [36] leverages monotonic counters to guarantee freshness in log files. They implement an *Asynchronous Monotonic Counter Interface* (*AMC*) to overcome the limitations of native SGX counters [38]. The motivation behind the AMC is to achieve overlapping between the period that is required for writing to a hardware counter and the time data needs to be stored persistently (many systems expose different persistence times [5, 2] or asynchronous commits [9, 11]).

AMC allows multiple increments of the asynchronous (software) counter overcoming, that way, the slowness of the synchronous (hardware) SGX-counters. The asynchronous counter's values are stored persistently in a file along with an SGX-counter value which is necessary to ensure the asynchronous counter's freshness.

As shown in Figure 2.2 we define the *unstable period* or *epoch* as the time distance between two synchronization points in the *timeline*. A *synchronization point* denotes that the current asynchronous counter has been stabilized (its value is stored persistently) and, therefore, cannot be reverted. Upon an increment, we are informed about the current *stable value*, the *incremented value* and the *time* this incremented value is expected to be stabilized. Typically, the unstable period equals to the time that a single SGX-counter increment operation requires to complete. According to several works [38, 35, 36] the unstable period varies from 60 ms to 250 ms depending on the platform.

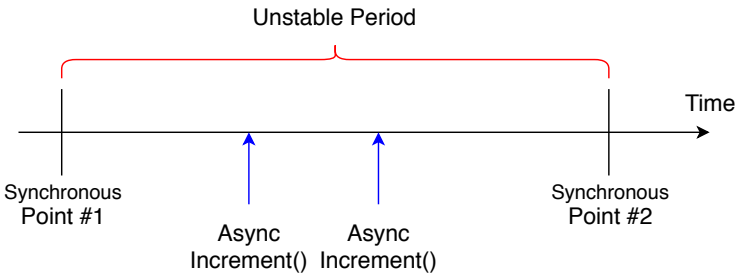

Figure 2.2: SPEICHER's Asynchronous Counter.

SPEICHER's AMC allows in theory the creation of an arbitrary number of software counters whose values are synchronized concurrently after the current unstable period passes. However, in practice, we can only have a limited number of them due to some interoperability issues.

Chapter 3

Protocols' Design

In this chapter, we first present a high overview of our proposed system (Section 3.1). In Section 3.2, we discuss the design of a naive protocol that achieves serializability and rollback protection leveraging the *native* SGX-counters [20, 7]. We analyse the limitations of such an approach that motivate us to use asynchronous counters, like those proposed by SPEICHER [36]. In Sections 3.4.1 and 3.4.2 we discuss our protocols' design for PCC and OCC, respectively. Lastly, in Section 3.5 we present the recovery algorithm.

3.1 Proposed System Overview

Figure 3.1 illustrates the overview of our system. We build our prototype on top of RocksDB and, in future parts of this project, we aim to integrate it into SPEICHER's infrastructure. SPEICHER decouples keys and values path in the MemTable and efficiently addresses the performance overheads due to the limited enclave memory. On top of this architecture, we propose transaction protocols that leverage the asynchronous counters to meaningfully serialize transactions' commit order while at the same time guarantee protection against rollback attacks.

RocksDB Engine runs inside the enclave thereby being isolated from OS and hypervisors. For the deployment, SPEICHER uses SCONE [17]. Each transaction locally buffers its updates to an individual per transaction in-memory data structure. RocksDB names a transaction's local buffer as `WriteBatch` and implements it as an `std::string`. In our first proposal, we keep these per transaction local buffers inside the enclave. We are aware that they may introduce contention for the enclave memory and, consequently, degrade the performance, however, we will address the memory

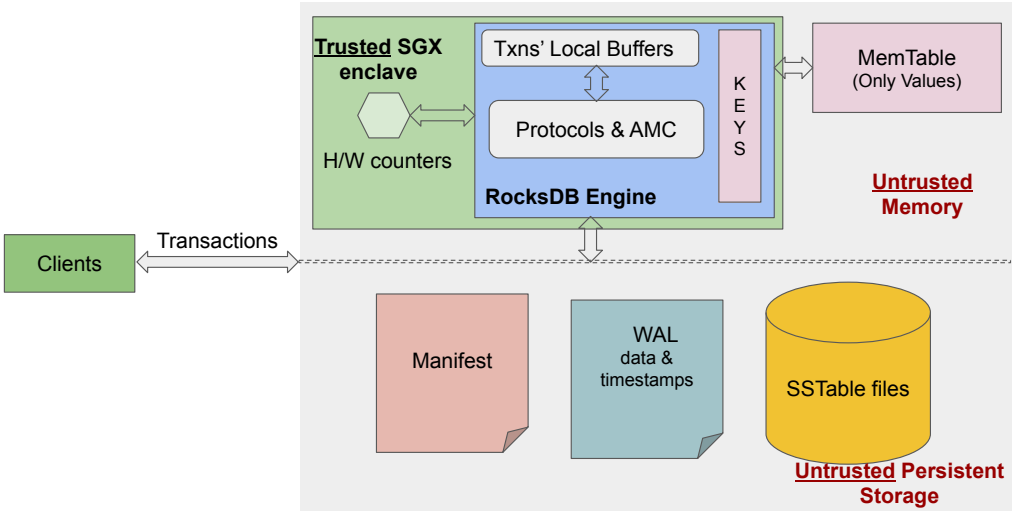

Figure 3.1: Proposed System Overview.

optimization in future directions of this project.

The core idea of our protocols targets to appropriately assign commit-timestamps to transactions and log these timestamps along with the data in the WAL. Timestamps determine the position of the transaction within transcactions serialization history. By using trusted counters for these timestamps our system is always aware of the latest committed timestamp prohibiting an adversary to replay stale data. Once a transaction's updates have been logged successfully and the timestamp has been stabilized, the transaction is irreversibly committed.

RocksDB's default configuration assumes that the updates are logged immediately, however, this option is configurable. RocksDB backend WAL-mechanism has already an internal buffer. For high durability RocksDB flushes the buffer each time a request for a write to the WAL is invoked, however, more relaxed durability semantics (e.g. asynchronous logging by postponing flushing the buffer until having some writes accumulated) could reduce the number of I/Os and decrease logging times. Indeed, other big data stores such as Cassandra [2] and HBase [5], also expose different persistence guarantee times. What is more, RocksDB merges/concatenates the updates from multiple transactions and writes this concatenated batch atomically to the WAL (*group commit*). In the same direction, employing parallel logging to a single or multiple log files could speed up logging times. That being said, our protocols rely on the assumption that the order of the transactions' entries in the WAL is not required to match their

commit order and thus, the timestamp ordering.

3.2 Using SGX Monotonic Counters

3.2.1 Naive Approach

To ensure resilience against rollback attacks and, at the same time, order the committed transactions meaningfully, a naive approach would be to *bind* each transaction with a unique id which denotes the transaction's commit-timestamp as shown in Figure 3.2. Having in our disposal the SGX-counters we define as a transaction's commit-timestamp the unique current value of the synchronous monotonic hardware counter. That way, the transactions' ordering is uniquely determined by the counter values.

Moreover, due to the nature of the synchronous SGX-counter, to guarantee rollback protection, we are obliged to follow an *inc-then-store* approach. At the commit phase, each transaction increments the hardware counter value. The returned value is considered to be the commit-timestamp of this specific transaction and it is stored persistently along with the data in the WAL.

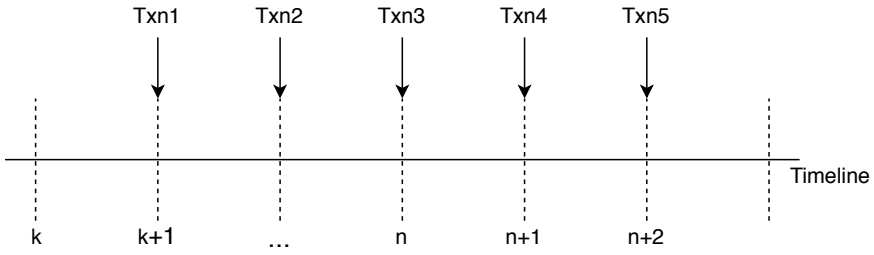

Figure 3.2: Transactions' timestamps equal to the SGX-counter's values.

Since the counter's returned value is stable, once the data are written in the log, they remain safe against rollback attacks. In case an adversary successfully manages to replay a stale log, the system can always identify this attack by ensuring that the replayed log, in case of a crash, contains transactions of the latest version. Equivalently, the hardware counter value matches with the latest transaction's timestamp. According to this, a successful recovery mechanism would apply the records of the WAL in the order denoted by their timestamps. However, if the system crashes after the increment operation and before the WAL's store procedure, the hardware counter will point to a future – not logged yet – state. Therefore, the mechanism would fail or it would be difficult to recover the system; the mismatch between the hardware counter value and

the latest logged timestamp would prohibit the mechanism from determining whether the current log is the latest one. Nevertheless, this approach guarantees us the strongest possible freshness attributes.

A sketch of our idea is shown below and it is common to both pessimistic and optimistic transactions:

Algorithm 1: Timestamp Assignment at the Commit Phase of a Transaction

```
1 /* Commit-phase. */
2 timestamp = atomicIncreaseSGXCounter(); /* can take 60-250msec */
3 /* Updates in the WAL and the MemTable are done in an atomic fashion. We
    use this unique timestamp to ensure the rollback protection and a meaningful
    serial ordering. */
4 CommitInternal(timestamp);
5 releaseResources();
6 return;
```

3.2.2 Limitations

This idea presents some severe limitations that make it as a whole impractical. First of all, the performance and the robustness of the SGX-counters are not very well documented [35, 36, 20]. Indeed, due to the fact that these counters use a non-volatile NVRAM area, several studies have shown [35, 38] that this memory area wears out after a few days of continuous use, rendering the SGX-counters unusable.

Apart from this, SGX-counters are relatively slow; a counter's update can take from 60 msec to 250 msec [35, 36] depending on the platform. In a high performant KV store, that continuously accepts transactions, such a delay is critical to the overall throughput [26]. Additionally, the increment operation itself remains a point of serialization since the transactions block until the ongoing transaction completes the increment procedure and acquires its timestamp. Given that an update operation can take up to 250 msec, using the SGX-counter interface would be a poor choice. Finally, as already mentioned, this approach does not provide *crash resilience*.

The source of the protocol's inefficiency is that it is over-conservative in determining the exact strict order of all transactions. Especially in cases where the ongoing transactions are *independent*, i.e. they update different key-value pairs, their relative order, as long as their updates have been stabilized, does not play any important role for the system's restoring process. For instance, in case of a system crash, the recover-

ing mechanism can apply the independent transactional updates reported in the log in any arbitrary order with any interleaving among them without leading the system to a faulty, inconsistent or stale state.

3.3 Using Asynchronous Monotonic Counters

Following the observation that a strict ordering is unnecessary for transactions that do not touch the same data, we can relax the serializability attribute and permit independent transactions to acquire the same counter value, the same timestamp, and be processed in parallel. This would increase the parallel efficiency and the overall throughput of the system without violating the consistency semantics of the recovered system. In other words, we can leverage the fact that serializability defines a partial order of transactions where conflicting operations only are ordered. Transactions that are not ordered in the partial order can be assigned the same commit-timestamp, e.g. for instance, they could be assigned the same counter's value. Relying on this, we use an asynchronous trusted counter, similarly to SPEICHER, that does not block until the increment value is stabilized. More over, to facilitate recovery we use a *store-then-inc* approach by defining the timestamp of a transaction to equal the current stable value plus one ($stableVal + 1$). Using solely the current stable counter value as timestamp will result in all transactions being seen as stabilized, which may complicate the recovery algorithm.

Moreover, we can still argue about the stability of the transactional updates. Transactions that commit during the n^{th} unstable period, will be rollback protected (stable) at the beginning of the $(n+1)^{th}$ unstable period, that is, after their timestamp will have been stabilized. As shown in Figure 3.3, our proposed idea allows multiple independent transactions to be processed in parallel without delays. Our protocols will only serialize the conflicting operations to a distinct unstable period enforcing *serializability in an unstable period granularity*. For instance, in Figure 3.3, *Txn1*, *Txn2* and *Txn3* are not conflicting, thereby, can be processed in parallel and acquire the same timestamp. In contrast, *Txn4* and *Txn5* are touching common data, therefore, our idea is to delay (*commit-wait*) the second transaction in order to commit to a subsequent epoch.

We choose to serialize transactions in an unstable period granularity in order to ensure that an operation's timestamp will have been first stabilized before another conflicting one is allowed to commit. If two conflicting transactions commit in the same unstable period then their commit order, since we are in an unstable period, is not

guaranteed to be preserved after a crash. This is our main reasoning for delaying transactions in order to ensure that past conflicting transactions are stabilized first. This approach is quite conservative but it exposes determinism in the system. For instance, if two or more conflicting operations are allowed to commit in the same unstable period and the system crashes before neither of them is stabilized, then during recovery the transactions are possible to be re-executed. However, the transactions' ordering may do not match their initial ordering (before the crash). Our reasoning solves this situation since, in this example, the later conflicting transaction will commit-wait, allowing the previous transaction's timestamp to be stabilized and also acquiring a timestamp of a subsequent unstable period. As a result, in case the system's crashes after the second's transaction commit but before its timestamp becomes stable, then, the ordering between these two conflicting operations can be determined (since the first transaction has already a stable timestamp) prohibiting, that way, an adversary from tampering with the conflicting transactions' ordering.

Especially, in distributed transactions, this delay can be overlapped with other operations, like the replicas synchronization. It is necessary to ensure that an update must have been stabilized in all replicas before any other conflicting transaction is processed. Otherwise, similar to previously, it is possible to allow an adversary to change the order of the transactions. Our idea is mainly inspired by Spanner [19] which forces operations to *commit-wait* in order all clocks to be synchronized with the wall-clock. This delay overlaps with the PAXOS consensus algorithm [33] to synchronize all replicas in the geo-distributed database and its impact to the performance can only be considerable when the uncertainty period (clock skew) is relatively big. Therefore, extending our protocols for distributed transactions, the idea of the *commit-wait* is not only to synchronize clocks but also offer determinism in terms that all replicas should have the same view of the stable transactions before subsequent conflicting operations proceed.

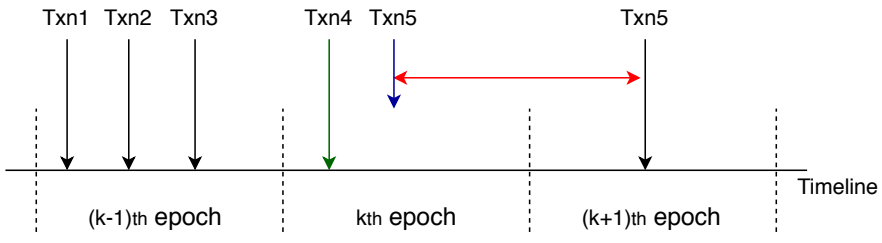

Figure 3.3: Serialization at *epochs* granularity. Conflicting transactions will be delayed to commit to a subsequent epoch.

3.4 General Approach

Our core idea to implement the serialization in an unstable-period granularity requires that every transaction, entering at the commit phase, should be aware whether there are conflicts with its keys in the current unstable period. To implement this functionality, the history of the committed updates in each unstable period is required to be known. We introduce an *Index*, an in-memory data structure which stores all keys that have been updated/written in the current unstable period, in other words, all keys that have been updated and persistently stored in the log but have not yet been stabilized. Once this period passes, this Index must be emptied (at synchronization points) so that any incoming transactions can successfully determine if there are conflicts at their commit-phase in the current unstable period.

As mentioned previously, we define the timestamp of a transaction to equal the current stable value plus one ($stableVal + 1$). All independent transactions acquire the same timestamp. After a transaction successfully commits, it increases the asynchronous counter value. In case of a system crash between logging the transaction and increasing the trusted counter, the transaction will have an non-stable timestamp, because its commit-timestamp will point to a future trusted counter value. Therefore, the recovery mechanism can simply discard it or notify the client to re-issue the operation. Since the client can be aware of when their updates will be stable, they need only to cache their transaction for such a period of time. After that, their transaction has been stabilized successfully and an adversary's attempt to rollback it can be detected.

3.4.1 Protocol for Pessimistic Concurrency Control

A sketch of our protocol when using the Pessimistic Concurrency Control, is shown in Algorithm 2. Every transaction that enters the commit phase will first search whether at least one of its keys already exists in the temporary Index. If conflicts have been found during this search, then that denotes that another transaction has written these keys in the database during the current unstable period. Therefore, the ongoing transaction should commit-wait for the current unstable period to pass in order to acquire a timestamp that matches to a different synchronous point; namely, to commit in a subsequent unstable period. After the transaction successfully commits, the Index must be updated with the transaction's keys so that any dependent transaction(s) can later identify any potential conflicts. Before releasing the locks of the keys and unblocking dependent transactions, this specific transaction should also increase the counter

value. That way, we ensure that the keys will be "live" in the Index at most for a single unstable period, since once the stable counter value advances, the Index will be cleared. Equivalently, that way, we ensure that for any conflicting transactions, a single commit-wait will guarantee commit-timestamps of different unstable periods.

Algorithm 2: Pessimistic Transaction Protocol

```
7  /* entering the commit-phase with the locks of all keys acquired */
8  conflictDetected = searchIndex(txnKeys);
9  if conflictDetected then
10 |   commitWait();
11 timestamp = getCounterStableValue() + 1;
12 success = CommitInternal(timestamp);
13 if success then
14 |   updateIndex(txnKeys);
15 |   increaseCounter();
16 ReleaseLocks();
```

The fact that we update the Index after the transaction has appended the updates to the store (WAL and MemTable) and before the transaction releases the locks ensures us that every dependent transaction will "see" the updates in the Index. If the updates are not present that means that the stable counter value has been proceeded and thus, it is ensured that the next dependent transaction will acquire as a timestamp a subsequent synchronous point. It is important to note here one corner case that we found interesting: Since the acquisition of the timestamp and the update of the Index are in a parallel context and not happen atomically it is possible that all or some keys are appended to an Index which matches to some subsequent unstable period. Consider, for instance, a very long transaction that takes too long to be appended to the WAL and the MemTable so in the meantime the unstable period has advanced, emptying the Index. However, even in this case, when a blocking transaction enters commit-phase and find some keys in the Index, it will conservatively commit-wait ensuring again serialization in an unstable period granularity since it will be assigned with a different, greater timestamp.

3.4.2 Protocol for Optimistic Concurrency Control

We remind that under the OCC transactions take no locks on their constituent keys and the conflict detection is done during the transaction's commit stage. The logging protocol elects a *writer thread*, which has atomic access to the database, and this thread is responsible for checking whether the keys it will update are of the latest version, namely, no other intermediate transactions have updated these keys (Appendix A.2). If no conflicts are detected, then the updates are written to the database and the commit is successful. Otherwise, the transaction is guaranteed to abort.

The challenge in OCC stems from the absence of locks, which in the previous case implicitly offer serialization; unless a transaction releases the lock on a key, no other conflicting transactions – that operate on the same key – can commit. Indeed, we took advantage of this mechanism to ensure that there is only a single transaction a time that writes a key to the Index since any depending transactions are blocked until the lock of this specific key is released. Therefore, taking no locks on keys makes the pessimistic protocol inapplicable. Towards the design of the optimistic protocol we come up with three possible solutions:

- The most obvious idea is to embed the search and the update operations of the Index inside the RocksDB logging protocol where concurrent writers are synchronized. That would ensure exclusive access to the Index; the writer thread which has the exclusive access to the WAL would also has access to the Index. However, in cases where a transaction should commit-wait, this operation will take place in the *critical path* delaying subsequent writer threads from logging their updates and therefore, raising performance concerns. Another disadvantage is that our protocol would not be modular since they would be highly depended on the current specific logging mechanism. For instance, if the underlying logging protocol exposes parallelism, as in works [27, 28], the serialization would be made difficult.

- Another approach is to use an Index that is protected by a global `mutex`. This would ensure atomic updates since the first transaction that would enter the commit phase, would also be the first to "lock" its keys. In case where another transaction found conflicts in the Index, it will have to commit-wait for the unstable period to pass. Again, the conflicts would indicate that another transaction had previously updated (or it is about to update) the same keys in the same unstable period. However, the use of a single global lock is a poor choice in terms

of performance because it does not guarantee any fairness – a transaction may systematically fail to acquire the lock – and minimizes parallelism.

- Our most promising idea is to implement an *Atomic Search and Update* (*ASU*) utility that supports atomic and parallel `append` operations. In particular, a single ASU operation atomically updates the Index with a key returning `true` in case the current key does not exist and `false` in case the key already exists in the Index. This atomic ASU operation makes sense to be issued at the beginning of the commit phase of each optimistic transaction for all transaction's keys. The idea is that transactions "lock" their keys in advance – in terms of updating the Index – so that other parallel transactions can identify any conflicts when entering their commit phase. Similarly to the pessimistic case if a transaction does not identify conflicts, then it can freely commit. Otherwise, the transaction should commit-wait for the unstable period to pass. The difference to the pessimistic protocol is that after the commit-wait a transaction is not necessarily eligible to proceed since due the absence of locks it must again upfront lock its keys to block other conflicting operations. In the case, keys' conflicts are still present in the Index, the transaction aborts. The reason that the keys are still locked implies that during the current transaction's commit-wait another more recent transaction reached the commit phase. Forcing the current transaction to abort is not orthogonal to the original protocol's semantics. Since a more recent transaction will commit prior to the stalled transaction, the later would identify conflicts at the validation phase. In other words, we expose a parallel validation phase that also ensures serialization in an unstable period granularity in the upper software layer.

Following the third approach, a question that needs to be answered is *when exactly the transaction should acquire its timestamp*. If we adopt the similar idea as the pessimistic case, we could implement a logic, as shown in Figure 3.4, where a transaction enters the commit phase, it upfrond locks its keys and in case of success it acquires a timestamp and commits its updates. However, our asynchronous counter acts independently, e.g. does not wait for a transaction to finish Index updating before proceeding to the next unstable period, and therefore, a problematic situation as also shown in Figure 3.4 could arise. In that corner case, the updates of the first transaction are vanished (because the Index is cleared every time we change the epoch) and therefore, these updates are invisible to the second transaction which might also acquire the same

timestamp. In the end, two conflicting transactions might appear in the log with the same timestamp which violate our serialization semantics.

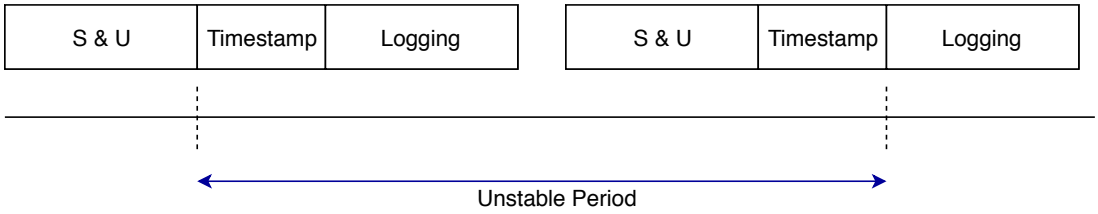

Figure 3.4: Bad Synchronization may cause wrong ordering of transactions.

Following an opposite approach, according to which a transaction first acquires the timestamp and then updates the Index, would guarantee that the transaction's updates would be visible to the current or may the subsequent unstable period thereby solving the previous problem. Unfortunately, an analogous situation could arise as shown in Figure 3.5.

Figure 3.5: Bad Synchronization may cause wrong ordering of transactions.

Both cases arise from the loose synchronization between asynchronous counters and transactions' *lifetime*, namely, the fact that the timestamp acquisition and the Index update are not guaranteed to happen atomically in the same unstable. As a result, an Index flushing may vanish important information about the past epoch's transactions. We could naively solve this problem by not emptying the Index at all. For instance, we could have a KV-based Index where key would be the transaction's key and the value the transaction's acquired timestamp. However, such an idea may do not offer fast lookups since the Index would become huge even with few thousands of keys. Another problem would be the memory requirements for such an implementation because the Index would continuously grow in an unpredictable way and thus, the already limited enclave memory would soon be saturated. Lastly, we could enforce a policy similar to Obladi [21], where each transaction is obliged to commit in the epoch it initially started. However, we want to keep the unstable period as small as possible and this approach in our system would be problematic especially for long transactions.

We overcome this problem by assigning a specific Index to each unstable period. We also effectively overcome the memory limitation by applying a *garbage-collection-based deletion policy* for the past indices. This idea solves the case presented in Figure 3.5 since both transactions would search and update the same Index which is determined by their timestamp. Also, this idea helps us keeping the indices relatively small, improving the lookup times. To efficiently use memory, we lazily delete past indices that are not useful anymore. In particular, every time the unstable period advances we create a new Index for this new period. At the same time, we delete any past indices that have not been accessed in the previous epoch since this condition implies that there is high change that they are not useful anymore. This deletion policy allows us to be adaptive to workloads with very long transactions that may require the respective Index to be *alive* for more than one unstable periods. A more strict deleting policy would be to delete the previous epoch's Index. However, that may cause unnecessary commit-waits or aborts in workloads with very long transactions, presuming that updating the Index with all keys may take more than a single unstable period to be completed.

Algorithms 3 and 4 show the pseudocode for the Optimistic Protocol and the Search and Update Utility, respectively.

The atomic nature of the ASU operation ensures us that if two or more transactions invoke this method with the same key only one transaction will be allowed to proceed (return `true`) while the remaining transactions will be forced to commit-wait. Lastly, we append the keys in the Index in a sorted fashion to avoid any deadlocks.

Algorithm 3: Optimistic Transaction Protocol

```
17 /* entering the commit-phase */
18 timestamp = getCounterStableValue() + 1;
19 conflictDetected = SearchAndUpdate(txnKeys, timestamp);
20 if conflictDetected then
21 |   commitWait();
22 |   timestamp = getCounterStableValue() + 1;
23 |   if SearchAndUpdate(txnKeys, timestamp) then
24 |   |   return Status::Busy();
25 success = CommitInternal(timestamp);
26 if success then
27 |   increaseCounter();
28 ReleaseResources();
```

The inspiration for this algorithm comes from the fact that Optimistic Concurrency Control is solely suitable for workloads with little write-write conflicts. Relying on this, we believe that appending the keys to the Index in advance, without knowing whether this transaction will abort, successfully commit or, even, commit-wait, will not act negatively to the overall system's performance, e.g. causing other transactions to commit-wait unnecessarily.

Algorithm 4: Search and Update Utility

```
29 /* SearchAndUpdate(sortedKeys, txnTimestamp) */
30 for key in sortedKeys do
31 |  /* the append operation is atomic and returns true if the Index is empty and
   |     the key has appended successfully and false if the key already exists */
32 |  if !ASU(key) then
33 |  |  return true; /* conflict found */
34 end
35 return false;
```

3.5 Recovery Algorithm

The role of a recovery mechanism with rollback detection is to replay the log in order to revert the database to its state before the crash, applying all recovered transactions that are stable and also identifying that the replayed log is of the most recent version, thereby non compromised by a potential rollback attack. Our restore algorithm takes into account the latest stable timestamp counter value which indicates the latest rollback protected epoch. In particular, all transactions in the WAL that have a timestamp value at most equal to the latest stable counter value are considered to be safe and rollback protected, therefore, they can freely be applied with respect to their timestamp order.

During the recovery, a record must have stable commit-timestamp guaranteeing that way rollback protection of the ordering for successfully committed transactions. Therefore, we discard all transactions with a timestamp greater than the latest stable value, since we cannot trust their ordering.

RocksDB's WAL stores all updates in *records* of variable size. In Section 4.2 we explain in detail the WAL format and how we adapt it to our logic. However, we point out this detail because that indicates that a transaction may be spread to multiple

records. Also since many transactions are allowed to have the same timestamp, multiple records – which may refer to one or multiple transactions – are possible to have the same commit-timestamp which prohibits our mechanism from ensuring the *state continuity* property. In other words, since the log is stored in an untrusted storage an adversary may delete or duplicate WAL entries and even through the latest transactions' freshness can be proved, the resulting recovered system may be inconsistent and incorrect.

Towards this direction we use a second asynchronous counter (*WAL counter*) to add a unique and monotonically increasing *id* to each transaction's `WriteBatch`. Similarly to SPEICHER [36], we follow this approach rather than enumerating each constituent record separately because in the latter case it might be possible a scenario where only some of the transaction's records are stable in the WAL[1]. This would made the recovery difficult since all transaction's records should have been stabilized in order to apply the transaction as a whole. If the transaction is fragmented and an adversary compromise some partial log records, RocksDB's corruption checking mechanism identifies that[2].

It is worth noting that the timestamp counter and the WAL counter act in different system's layers (Figure 3.6). They are logically designed to serve two different roles; the WAL counter ensures us that no updates are missing in the log history and proves WAL's freshness, while the timestamps inform us about their ordering. In the current work, we do not collapse the two counters because, as part of our design, we conservatively chose to preserve the information about transactions' stability (with the timestamp counter) to all software layers. Given that we do not rely on RocksDB's logging protocol for serialization, using only the WAL counter would be insufficient to argue about the ordering of transactions.

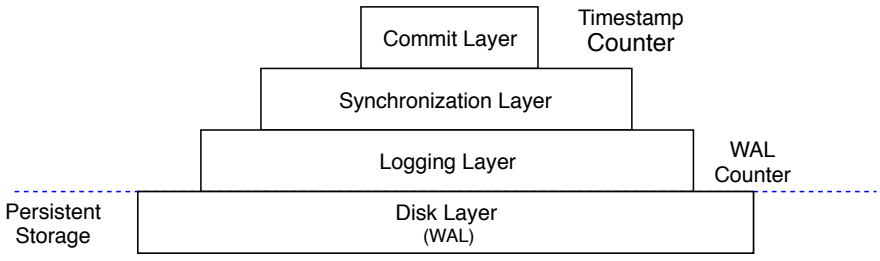

Figure 3.6: Logical Layers' where the trusted counters "live".

Algorithm 5 shows the recovery algorithm with rollback detection guarantees. We iterate sequentially over all WAL records checking for the deterministic increment of

[1] A transaction that has not stable *WALid* should be discarded during recovery.
[2] RocksDB uses check-summing and *sequence ids* to detect corrupted entries.

the log entries' ids. Particularly, the recovery process reconstructs each transaction and accepts all transactions that are stable in the WAL (line 37). Transactions whose stability cannot be proved will be discarded. We need to ensure that no transactions are missing or are duplicate so we check for the deterministic increment of the WAL counter (line 38)[3]. Afterwards, we only accept the transactions whose ordering can be determined, namely, whose trusted timestamp counter value has been stabilized (line 39). Before applying the transactions with respect to their order, we need to prove the WAL's freshness (line 45), that is, the latest log entry has WALid that matches the stable WAL counter value. If the check passes, we can freely apply all valid transactions.

Algorithm 5: Recovery Algorithm

```
36 for every txnWriteBatch in WAL do
37     if WALId is stable then
38         if WALId == prevWALId + 1 then
39             if stableTimestamp then
40                 orderedTxns = serialize(txnWriteBatch);
41         else
42             return corrupted_log
43         end
44 end
45 if WALId != WALStableVal then
46     return stale_log;
47 for every txn in orderedTxns do
48     apply(txn);
49 end
```

[3]The WAL's initial record id is assumed to be retrieved by a rollback-protected file like SPEICHER's Manifest.

Chapter 4

Implementation Details

4.1 Index data structure

In the previous Chapter, we introduced an in-memory data structure, the *Index*, that it was treated as a black-box. Its role is to keep all epoch's live keys, namely, all keys that have been updated by some transaction in the current unstable period. As discussed, searching and inserting keys in this Index is crucial in order to force the conflicting transactions' commits in the proper unstable period. In this Section we present our analysis regarding the implementation of this Index.

4.1.1 Implementation Challenges

In general, our system is designed to support multiple clients, so the Index is required to fulfill the following two properties:

- It must support fast and concurrent lookups because any ongoing transactions should be able to search the Index in parallel and, relatively, fast.
- It must also support concurrent and thread-safe insertions because independent transactions should not be blocked while appending their keys to the Index.

4.1.2 Hash Map with *SWMR* locking mechanism

Our first prototype has been developed with a hash map wrapped by a *Single-Writer-Multiple-Readers* (*SWMR*) locking mechanism. The average lookup complexity for a hash map is *O(1)* and in combination with the allowance for parallel readers to be active, our prototype served the requirement for fast lookups. However, allowing only

a single writer a time to be active minimizes parallelism for insert operations which make it unsuitable for a system with many parallel clients as ours. This idea was early abandoned.

4.1.3 Skip List for Pessimistic Transactions

Our next idea is to use a *Skip List* [40], a probabilistic alternative to a balanced tree that works efficiently when inserting keys in random order. More specifically, we are motivated by the Skip List implementation that it is already provided by RocksDB library (`inlineskiplist.h`) to implement the MemTable. RocksDB's Skip List is highly optimized for fast lookups and more importantly, it supports concurrent accesses (both insertions and searches). The greatest advantage is that these concurrent accesses are allowed without any external synchronization (from programmer's perspective). The only restriction is that insertions are legal as long as they are not invoked in parallel with other keys that are compared as equal.

The properties of this Skip List match well to our requirements, especially under Pessimistic Concurrency Control. The restriction that discussed above is not a problem in pessimistic transactions. The locking mechanism serializes implicitly the commit-phases of conflicting transactions so no equal keys are concurrently appended to the Skip List.

To sort the keys in the Skip List we also make use of RocksDB's default comparator (`Comparator::BytewiseComparator`) which implements bytewise comparisons among the keys thereby making our implementation key-type agnostic. Our implementation however is modular; we can implement the `Interface` of RocksDB comparator to provide our own costum comparators. Lastly, we also use a thread-safe *memory allocator* (`Allocator::ConcurrentArena`) to allow concurrent creation of new Skip List nodes. One last remark is that we only make use of the Skip List's key-path. That way, we expose limited memory requirements since the default key size is 16B[1].

4.1.4 Concurrent Hash Map for Optimistic Transactions

The use of the Skip List presents some major advantages the most important of which is the lack of the need for external synchronization while still offering concurrency. However, this specific version of a Skip List is impractical for transactions under the Optimistic Concurrency Control. The problems arise from *a.* the fact that the Skip

[1] In general, a key can have an arbitrary size, however, RocksDB is not optimized for very large keys.

List's `append` and `search` operations are not atomic[2], and *b.* the nature of the optimistic transactions which take no locks for the keys. Combining these two aspects, we cannot implement our proposed ASU operation with a Skip List.

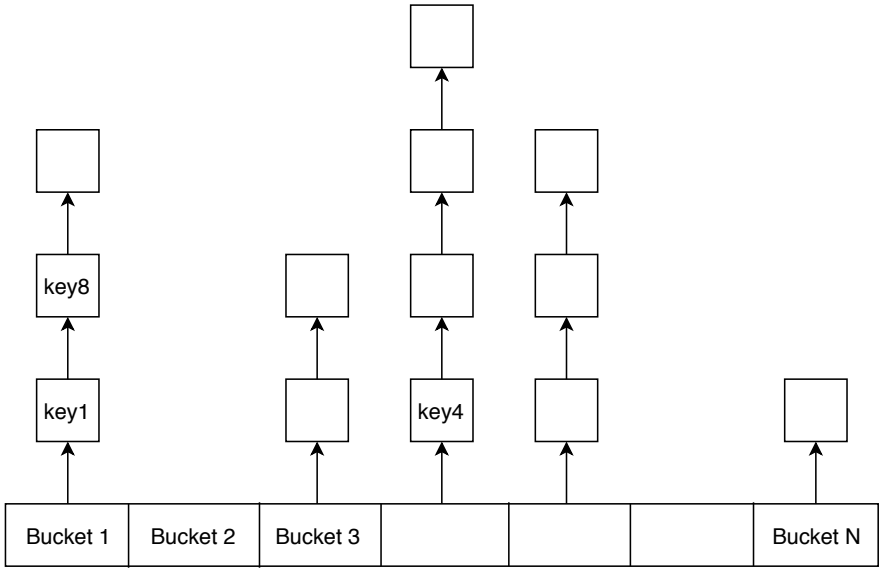

Figure 4.1: Concurrent Hash Map with Linked Lists.

To tackle this problem we design a *concurrent thread-safe Hash Map* that supports concurrent lookups and exposes some form of parallelism while appending new keys. The sketch of our structure is shown in Figure 4.1. The Hash Map associates a *bucket*, with a single, not sorted, linked list. The distribution of the keys to the buckets mainly depends on the hash value of keys and the Hash Map's size.

To calculate the bucket to which a key will be assigned we used the following formula: $bucket = hashFn(\texttt{key}) \bmod hashSize$, where the *hashSize* is equal to the size of the Hash Map, equivalently, the number of the buckets (configurable) and the *hashFn* is the `std::hash` hash function [13]. To further allow some form of concurrency, we use a per-bucket `mutex` and consequently, our Hash Map allows concurrent appends and lookups as long as the keys match to different buckets. Lastly, to achieve a finer keys' distribution and thus, enhance parallelism, we also choose a prime and relatively big number of buckets. In our implementation we configured the hashSize to be equal to 1001 and our experiments showed minimal overhead in parallel optimistic transactions.

[2]If two parallel transactions call the `append` operation *concurrently* for the same key it is not guaranteed that only one of them will succeed. Additionally, taking no locks allows the concurrent invocation of the `append` function for the same key by two distinct transactions.

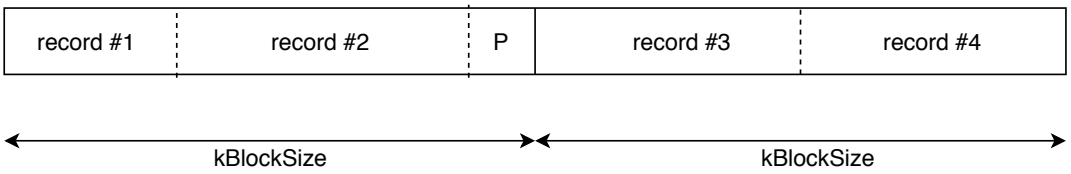

Figure 4.2: WAL Format.

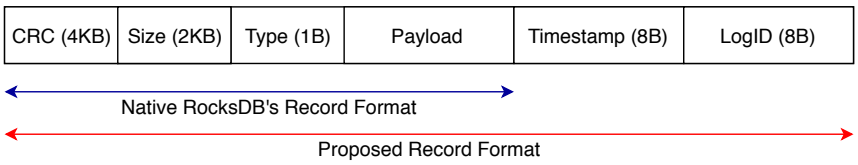

Figure 4.3: WAL Record Proposed Format for state continuity and ordering.

4.2 Write-Ahead-Log Format

As shown in Figure 4.2, RocksDB's WAL file is organized into *physical blocks* of fixed size (32KiB). A physical block consists of one of multiple *records* of variable size. Each record has a header (7B) which consists of a *checksum* of the following *payload* (byte stream), the size of the payload and the type of the record (`full`, `first`, `middle`, `last`). Right after that, it contains the data as a plain byte stream. If the block's remaining space is less or equal to the header size, it is padded and the WAL's pointer advances to the next block. Therefore, in block boundaries or when logging long transactions, a transaction may be spread into multiple records and physical blocks.

To identify the commit-timestamp of each transaction and also assign a log Id to the logged transaction (for WAL's freshness) during recovery we store the timestamp along with the WAL id to the latest transaction's record (`full` or `last` type) as shown Figure 4.3. As a result, our implementation safeguards against adversaries who they want to replay an older WAL, reverting the system to a previous state, or even a log with reordered entries – the timestamps always guarantee us the proper order of the committed transactions. Similarly the deterministic WAL counter's increment ensures us that no missing or duplicate entries exist.

Chapter 5

Performance Evaluation

5.1 Methodology for Evaluation

For the performance evaluation, we use the `dbbench` benchmark [4] which is a rewrite of Google LevelDB's [8] benchmarking tool. The benchmark is compiled along with the RocksDB library and the header files and offers homogeneous deployment (one binary). Our execution scheme takes advantage of this binding to extensively stress-test the system. For instance, in a server-client architecture, IPC or network latencies may not always bring the system under full workload conditions.

We define the *throughput* (MB/sec) reported by the `dbbench` as the performance metric of our evaluation. In a multi-client execution throughput equals the accumulated MB per thread divided by the actual elapsed time (not the sum of per-thread elapsed times). `dbbench` uses a random key distribution based on a Xorshift [34] generator.

5.2 Experimental Setup and Workloads

We run our experiments in a server with Intel(R) Xeon(R) CPU E7-8857 v2 (3.00GHz, 4 sockets, 48 cpus, 96 hyper-threads). Each cpu has 32KB for L1d and L1i caches, 256KB L2 and 30MB L3 caches. The server has also 1.48TB main memory and 2.2 TB disk space. We developed RocksDB `v5.18` and all software is compiled with `gcc version 6.3` with the default flags provided in the `Makefile` (`make static_lib`).

We evaluate our system for different parameters. First, we introduce two workloads, *Workload A*, which consists of *small* transactions (10 `PUT`s/Txn) and *Workload B*, which issues *longer* transactions (100 `PUT`s/Txn). The number of total transactions in both workloads remains fixed across our runs (see Tables below) and is equally di-

vided among the threads. Our reasoning is to keep the total amount of transactions fixed and examine the system's scaling capabilities (1, 2, 4 and 8 clients) and to evaluate the impact of the commit-wait operations w.r.t. a native – without security semantics – RocksDB instance. Additionally, we test our system for variable transaction's sizes or *batch sizes* – that is the number of `PUT`s/Txn – to acquire a better understanding on the impact of the commit-waits. In particular, we test for batch sizes to be equal to 500, 1000 and 2000 `PUT`s/Txn which lead to workloads with 50M (50GB), 100M (100GB) and 200M (200GB) KV pairs, respectively.

The `key` and `value` sizes are 16B and 1024B, respectively (default sizes).

Workload A				
#Txns	PUTs/Txn	Key Size	Value Size	#KV
100K	10	16B (default)	1024B (default)	1M (1GB)
Workload B				
#Txns	PUTs/Txn	Key Size	Value Size	#KV
100K	100	16B (default)	1024B (default)	10M (10GB)

In our experiments we configure the unstable period to 60 ms which, according to previous works [36, 38, 35], is the minimum amount of time an SGX-counter's write requires to complete. To construct synthetic workloads that expose different percentages of conflicting operations, we restrict the range of the randomly generated keys using the function: $key = randomGenerator() \ \% \ RANGE$. During the experiments, we collect statistics about the number of the issued commit-waits, the successful and the aborted transactions. Therefore, in our experiments we define $x\%$ of conflicts as the $\frac{\text{commit waits}}{\text{total transactions}}$. The value of the *RANGE* that fits in each of our scenario has been found experimentally. All measurements have been calculated as the average of 5 to 10 runs.

5.3 Results and Analysis

5.3.1 Native RocksDB Analysis

Figures 5.1 and 5.2 show the throughput of a native – without security guarantees – RocksDB instance under Workloads A and B for both pessimistic and optimistic transactions. In this system configuration, we do not interfere with the benchmark's key generator and the operations barely present conflicts.

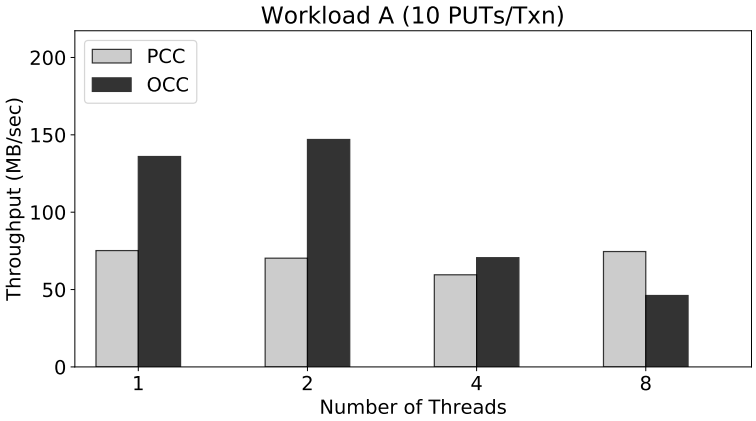

Figure 5.1: Throughput of native RocksDB with PCC and OCC under Workload A.

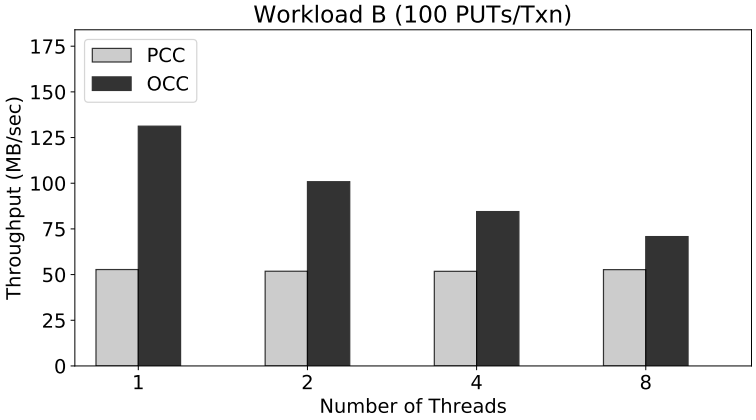

Figure 5.2: Throughput of native RocksDB with PCC and OCC under Workload B.

Two are the key observations. First, the PPC offers little or no scalability while the OCC scales badly with the number of clients. The first aspect is probably attributed to the contention in the *Key Lock Manager* and to the non-scalable RocksDB's logging protocol (only one write thread a time can log updates). Moreover, the contention for shared resources and the HDD drives might introduce latencies for both PCC and OCC. Lastly, given that RocksDB has also other background threads for compaction (2 by default) NUMA effects (e.g. NUMA *affinity*, first touch policy [30]) may affect the overall performance. In our platform each NUMA node only consists of 8 cpus, therefore, when having many clients the possibility a thread to be scheduled to another NUMA node is increased.

This analysis also reveals the PCC's locks' overhead. Indeed, optimistic transactions can achieve up to 2$\times$ higher throughput. In contrast, we see that OCC can be a poor choice with many concurrent clients (8 threads in Workload A) and, additionally,

it scales badly. The reason is that the system for OCC performs conflict checking only at commit time in a serial fashion disabling group commits (Appendix A.2). Consequently, all active threads must be synchronized and serially each of them will validate its write set. The synchronization cost in combination with the validation phase and the multiple small sequential I/Os lead to a lengthy commit phase for OCC and threads' stalls. This is not present in PCC since locks offer parallel validation alleviating such overheads during commit.

5.3.2 Protocols with Synchronous SGX-Counters

We evaluated the system's performance under Workloads A and B when using the SGX-counter based protocol (Section 3.2). Our results showed that the performance suffers irreparably for both workloads (240× worse performance on average w.r.t. native RocksDB) and strengthen our decision to abandon this naive protocol.

5.3.3 PCC Protocol using the Asynchronous Counter

Figures 5.3 and 5.5 depict the slowdown of the system with the PCC protocol w.r.t. the native's RocksDB throughput for Workloads A and B, respectively. Both workloads enjoy similar to the native performance when the commit-wait action is deactivated (red line), therefore, our structure's choice enhances parallelism and is not a bottleneck itself.

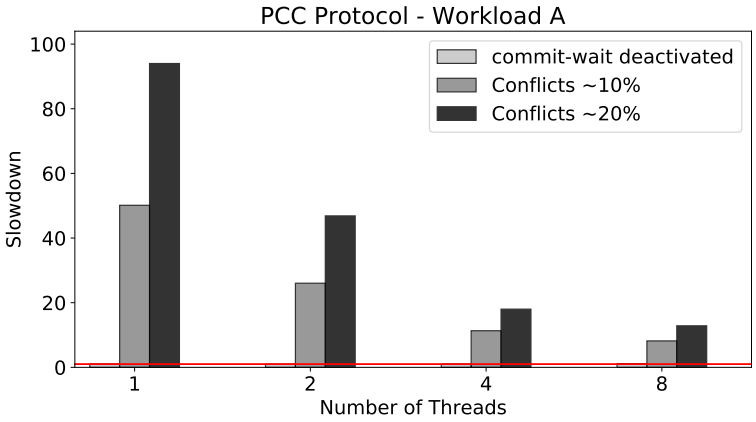

Figure 5.3: Slowdown of PCC Protocol w.r.t. a native RocksDB instance under Workload A.

However, even with the asynchronous counters, the overhead for 10% and 20% conflicts in Workload A is high. The reason is that small transactions are logged quite

fast so there is not overlapping between the times when writing to the database and the commit-wait of parallel transactions. As a result, even with moderate conflicts (10%), the system's stalls introduce great overhead. On the other hand, as shown in Figure 5.4, PCC protocol for Workload A with 20% conflicts offers 4× (run with 1 thread) to 30× (run with 8 threads) better performance compared to when using the SGX-counter.

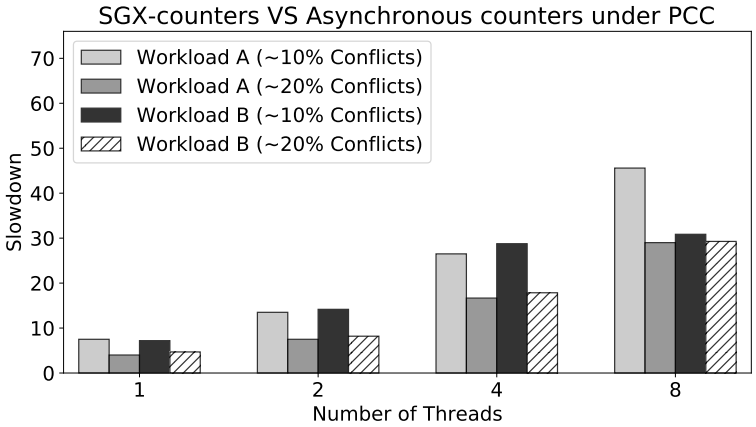

Figure 5.4: Slowdown of SGX-counter Protocol w.r.t. PCC Protocol (with asynchronous counters).

In contrast to small transactions, when comparing our PCC protocol with a native RocksDB's instance, we observe reasonable (13×) to minimal (1.4×) overheads at Workload B (Figure 5.5). Even with 20% conflicts, PCC offers 4.6× (run with 1 thread) to 29× (run with 8 threads) better performance compared to the protocol with the SGX-counter.

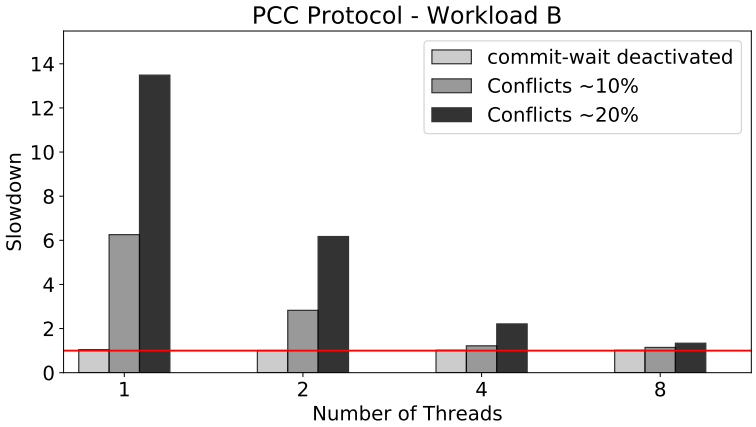

Figure 5.5: Slowdown of PCC Protocol w.r.t. a native RocksDB instance under Workload B.

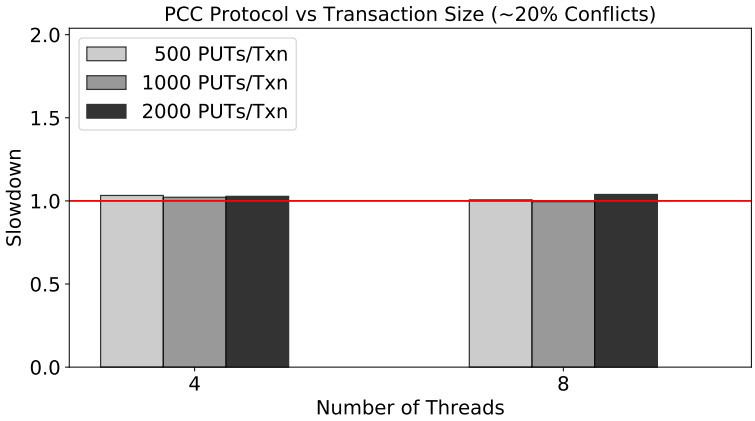

Figure 5.6: Slowdown of PCC Protocol w.r.t. a native RocksDB instance with varying batch sizes.

Figure 5.6 shows our results when testing PCC protocol with very long ($\geq$ 500 `PUTs`/Txn) transactions and many clients. We observe minimal or no overhead for such workloads. We attribute this to the overlapping between the logging time and the commit-wait operations of parallel independent transactions that eliminate stalls. Logging time is increased with long transactions so even though many clients are commit-waiting the system does not stall. A general observation is that our PCC protocol scales quite good with the number of clients in all type of workloads tested.

5.3.4 OCC Protocol using the Asynchronous Counter

Figures 5.7 and 5.9 depict the relative slowdown of OCC protocol for Workloads A and B, respectively. Similarly to the pessimistic case, OCC introduces high overheads at relatively small transactions. However, even such, we achieve up to 70$\times$ (run with 8 threads) better throughput compared to when using SGX-counters (Figure 5.8).

Moreover, Figure 5.10 proves that our OCC protocol's performance is almost comparable to the native RocksDB's performance at workloads with longer transactions even with a relatively high number of conflicting transactions ($\sim$ 10%)[1].

Additionally to the performance degradation, our experiments with the OCC show that an increment to the number of conflicts and threads deteriorates the system's ability to serve transactions since the number of aborted transactions is increased. We measured that the 5% of total transactions aborts when testing the system with Workloads A and B. However, in experiments with longer batches the system fails to serve

[1] As mentioned, OCC is a poor choice for heavier conflicts since many transactions eventually abort.

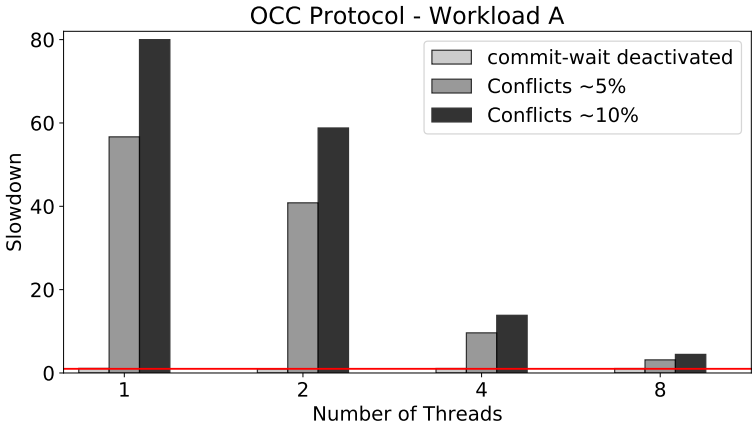

Figure 5.7: Slowdown of OCC Protocol w.r.t. a native RocksDB instance under Workload A.

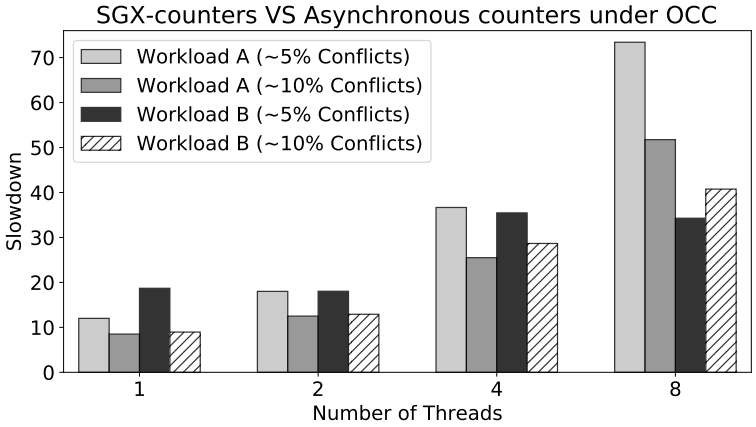

Figure 5.8: Slowdown of SGX-counter Protocol w.r.t. OCC Protocol (with asynchronous counters).

the 15% to 30% (Appendix C.1) of transactions. As expected, more parallelism in combination with a bigger batch size increased the chances for more conflicts leading more transactions to abort. It is worth saying that native RocksDB also suffers from a similar count of aborted transactions when issuing very long transactions. The fact that we observe more aborted transactions than the measured conflicts (commit-waits) is not something unexpected. We always measure the number of the successfully committed transactions that actually do a commit wait. However, it is possible to have many more “actual” conflicts.

The random key generator is one of the main disadvantages of our benchmark since this random key distribution may not be representative to real-world workloads. One crucial and future extension of the current project is the evaluation with other

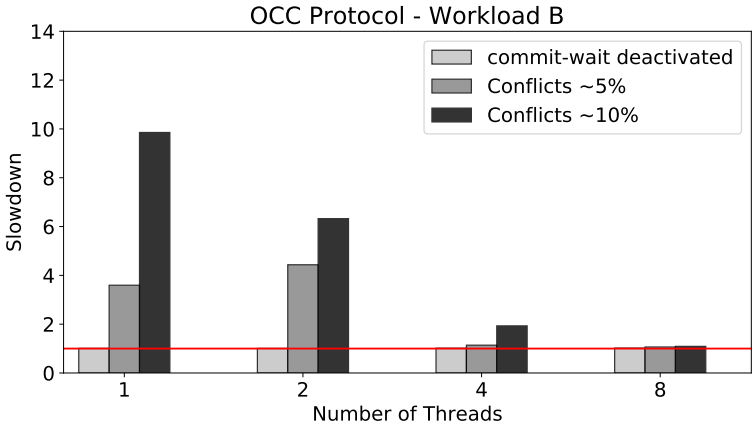

Figure 5.9: Slowdown of OCC Protocol w.r.t. a native RocksDB instance under Workload B.

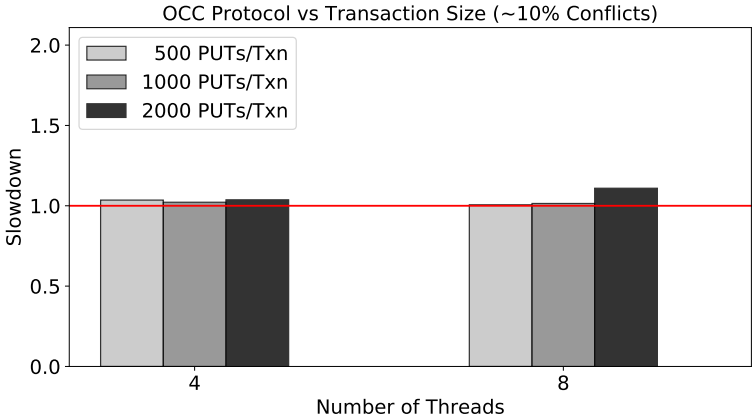

Figure 5.10: Slowdown of OCC Protocol w.r.t. a native RocksDB instance with varying batch sizes.

benchmarks [14]. Even though some well-known benchmarks, such as TPCC [14], have been studied, since they require a server-client architecture, their study was abandoned. Inter-process communication through sockets that is, for instance, required by TPCC is not efficiently handled by SCONE and SGX-architecture.

Chapter 6

Conclusion

6.1 Summary

In this project we investigated the design and the implementation challenges of protocols for Pessimistic and Optimistic Concurrency Controls that ensure serializability and rollback protection. We prove that SGX-counters degrade performance unbearably making the use of asynchronous counters irreplaceable. Our protocols combine a timestamp trusted counter and a WAL trusted counter to assign commit-timestamps to transactions, ensure rollback protection and detect adversaries who may reorder arbitrarily the records in the log, delete or duplicate them. Additionally, we design and implement a scalable and thread-safe Index that is optimized for concurrent accesses and fast lookups. The Index preserves the history of the past transactions' commits and it is used to achieve serializability in an unstable period granularity in order to ensure determinism in the order of successfully committed transactions.

Our evaluation shows that the use of asynchronous counters in general increases performance compared to when using SGX-counters. We noticed that our protocols fit better, in terms of performance, to workloads with long transactions and high concurrency because we achieve overlapping between the issued commit-wait operations and the time needed to update the database. Indeed, our system, tested under long transactions, enjoys almost similar performance for both pessimistic and optimistic transactions when compared to a native RocksDB instance without rollback resilience. For smaller transactions the performance's slowdown may be significant, however, even such, we achieve at least $4\times$ better performance compared to when using the SGX-counters.

We developed the algorithms in a single node system, but, by assuming only PCC

and OCC, the principles can be also generalized to a distributed context as well. The commit-wait operation ensures that all replicas are synchronized and also have the same stable state. In our case, the commit-wait operation offers determinism in terms that after a crash the system is able to preserve the transactions' initial ordering. This determinism however comes with a cost in performance for small transactions. As a future direction we could relax the requirement for determinism during the last unstable period. That way, we can allow multiple transactions to be committed in the same period avoiding commit-waiting and as a result, minimizing the expensive system stalls.

6.2 Future Work

6.2.1 Data Integrity and Confidentiality

We aim to harden the system's security guarantees by safeguarding data integrity and confidentiality. Particularly, we will leverage SPEICHER project which provides us with the building blocks towards this direction. As described in Section 3.1, our initial attempt will be to integrate these algorithms into SPEICHER. However, this approach can only be viable with small transactions given the limited enclave memory. To tackle with this problem we aim to investigate two possible directions: *a.* altering transactions' local buffers or *b.* exploiting other write policies in PCC.

Regarding the first approach, a promising idea would be to re-design transactions' local buffers by adopting a similar to SPEICHER's MemTable separation scheme, where only keys or meta-data of the local uncommitted batches are inside the enclave. This approach would be generic for both pessimistic and optimistic transactions.

6.2.2 Other Write Policies Investigation

To tackle with the limited enclave memory the investigation of the other write policies (Appendix A.1) in PCC could be also attempted. Their design is not orthogonal to our PCC protocol, except for the WAL records' format that should be revised. The notion of the commit-timestamp can be applied to these transactions as well, since they are implemented under PCC. The advantages of these policies are that they update the database ahead of time (before transactions' commit phases) and therefore, they keep minimal data in buffers (locally).

Nevertheless, they present their own challenges as well. For instance, an extra, in-memory data structure (*CommitCache*) is used to distinguish any non committed data that should not be visible. Lastly, OCC does not implement such write policies.

6.2.3 Parallel Logging

Another interesting question that has been arisen during this project is related to the efficiency of the logging mechanism since RocksDB currently does not support parallel logging (Appendix A.2). All concurrent writers are synchronized and only one of them, who has exclusive access to the database, serializes updates to the WAL. Although having a single thread may reduce the expensive I/Os in case the group commit is allowed, it can also delay logging time – especially when the batch is quite long. Parallel logging does not violate the overall system's architecture; we can still have the writes to the WAL and MemTable in a *pipelined* fashion, but each thread will be responsible for its own updates.

The challenges of parallel logging mainly arise in the OCC and MemTable's flushing operation. Locks in the PCC resolve conflicts. On the other hand, in optimistic transactions the conflict detection is done in a sequential fashion right before logging. To expose, therefore, parallelism we should also design a OCC parallel conflict detection mechanism. A type of this has already implemented as part of our serialization protocol in the optimistic case. Regarding the MemTable's flushing, currently the thread that gains exclusive access to the database schedules flushing. That ensures that no partial updates are written to the disk since all other writers are blocked. It is quite challenging to ensure this property in a parallel context.

6.2.4 WAL ordering for serialization

Our protocols assume that the WAL ordering is not required to respect the commit order. However, given the sequential RocksDB logging protocol we could "downgrade" the serialization to the WAL. That would simplify our protocols and would collapse the two counters since the WAL ordering would match the commit order. It is worth saying here that this direction contradicts the parallel logging we mentioned before since we rely on the current conservative protocol to serialize transactions.

6.2.5 Benchmarking

Well-known benchmarks such as TPCC [14] simulate appropriately real-world workloads. We aim to evaluate our implementation with them. We initially investigated their use, however, they require a server-client – typically through sockets – architecture which is detrimental for our future steps since SGX-architectures does not efficiently handle such operations. A very interesting and necessary contribution is to properly re-design and implement the system's architecture for using these benchmarks with SGX.

A high level approach of this architecture would be the use of a shared untrusted memory area[1] where the traffic generator process stores data, e.g. transactions. The application inside the enclave can continuously check this shared memory for incoming requests. Since the data are generated outside the enclave, their security properties should also be ensured. To achieve secure data exchange between these two processes, variations of *asymmetric encryption* [20] can be used.

[1] Applications inside the enclave have full access to the untrusted area.

Bibliography

[1] Arm trustzone. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed: August 12, 2019.

[2] Cassandra. http://cassandra.apache.org/. Accessed: August 12, 2019.

[3] Cockroachdb. https://www.cockroachlabs.com/. Accessed: August 12, 2019.

[4] dbbench. https://github.com/LMDB/dbbench/. Accessed: August 12, 2019.

[5] Hbase. https://hbase.apache.org/. Accessed: August 12, 2019.

[6] Intel sgx. https://software.intel.com/en-us/sgx. Accessed: August 12, 2019.

[7] Intel sgx documentation: create sgx monotonic counter. https://software.intel.com/en-us/sgx/sdk. Accessed: August 12, 2019.

[8] Leveldb. https://github.com/google/leveldb. Accessed: August 12, 2019.

[9] Oracle: Asynchronous commit. https://docs.oracle.com/cd/B19306_01/B14251_01/adfns_sqlproc.htm#ADFNS1018. Accessed: August 12, 2019.

[10] Overview of intel sgx - part 1, sgx internals. https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html. Accessed: August 12, 2019.

[11] Postgresql: Asynchronous commit. https://www.postgresql.org/docs/8.3/wal-async-commit.html. Accessed: August 12, 2019.

[12] Rocksdb. https://rocksdb.org/. Accessed: August 12, 2019.

[13] std::hash. https://en.cppreference.com/w/cpp/utility/hash. Accessed: August 12, 2019.

[14] tcpp-benchmark. `https://github.com/MinervaDB/tpcc-benchmark`. Accessed: August 12, 2019.

[15] Trusted computing group: Tpm 1.2 main specification. `https://trustedcomputinggroup.org/resource/tpm-main-specification/`. Accessed: August 12, 2019.

[16] A tutorial of rocksdb sst formats. `https://github.com/facebook/rocksdb/wiki/A-Tutorial-of-RocksDB-SST-formats`. Accessed: August 12, 2019.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, 2016. USENIX Association.

[18] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, 2014. USENIX Association.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[20] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[21] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 727–743, Carlsbad, CA, 2018. USENIX Association.

[22] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[23] David Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[24] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 7:1–7:14, New York, NY, USA, 2014. ACM.

[25] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[26] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A network programming interface for non-volatile main memory. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 17–33, Renton, WA, 2018. USENIX Association.

[27] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, September 2010.

[28] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal*, 21(2):239–263, April 2012.

[29] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 14:1–14:15, New York, NY, USA, 2019. ACM.

[30] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[31] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[33] Leslie Lamport. Paxos made simple. *Sigact News - SIGACT*, 32, 01 2001.

[34] George Marsaglia. Xorshift rngs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.

[35] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, 2017. USENIX Association.

[36] J"org Thalheim Maurice Bailleu and Michio Honda Kapil Vaswani Pramod Bhatotia, Christof Fetzer. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, Boston, MA, 2019. USENIX Association.

[37] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, New York, NY, USA, 2017. ACM.

[38] Bryan Parno, Jay Lorch, John (JD) Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, May 2011.

[39] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 264–278, 2018.

[40] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[41] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, Berkeley, CA, USA, 2009. USENIX Association.

[42] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, May 2015.

[43] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, Feb 1997.

[44] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

Appendix A

RocksDB System

A.1 Write Policies in Pessimistic Concurrency Control

A.1.1 WritePrepared

As discussed, RocksDB supports both Pessimistic and Optimistic Concurrency Controls. Pessimistic transactions make use of locks to achieve isolation between conflicting transactions and consequently they provide parallel conflict validation and concurrent processing of independent operations. PCC's default write policy is *WriteCommitted*, according to which data is written to the database, i.e., the log file and the MemTable, only after the transaction commits (client invokes `::Commit()`). This policy simplifies the read path because data seen by other transactions is assumed to be committed.

However, due to the fact that all writes are buffered in memory in the meanwhile, there have been identified some limitations in throughput and transaction size. For instance, memory can be a bottleneck – especially for large transactions. Also WriteCommitted generally leads to lengthy commit phases. Given that RocksDB only performs sequential writes to the WAL despite the concurrent transactions, lengthy commits can degrade throughput significantly.

To tackle with these issues RocksDB provides a *Two-Phase-Commit* (*2PC*) protocol – which the client must *activate* ahead of time. Relying on this, it implements two other write polices, *WritePrepared* and *WriteUnprepared*. Their core principle is to allow writing to the WAL and the MemTable at 2PC's earlier phases and thus, make the commit phase lightweight and fast.

2PC is composed of the *write stage* (`::Put()` is invoked), the *prepare phase*

(`::Prepare()` is invoked), and the *commit phase* (`::Commit()` is invoked) where the transaction's writes become visible to other threads. Therefore, to alleviate the overhead of the commit phase, WAL and MemTable's writes could be done at either *prepare* or *write* phases, resulting into WritePrepared and WriteUnprepared write policies, respectively. The challenge is to ensure valid reads on behalf of other transactions. For instance, when another transaction reads some data, it must know which data is committed.

With the WritePrepared policy, transactions still buffer writes in a `WriteBatch` in memory. At prepare phase, it writes the batch to the WAL and the MemTable along with a transaction's identifier (*prepare_seq*). At commit, the transaction writes a *commit marker*[1] to the WAL, whose *sequence number* (*commit_seq*), is used as the transaction's commit-timestamp. Of course, in the meantime many other transactions or query operations can be appended in the WAL. Therefore, the commit marker maybe part of a different group commit or even be stored in a subsequent log file.

Before allowing data to be externally visible, atomically with logging, the transaction stores a mapping from prepare_seq to commit_seq in the *CommitCache*[2]. When a transaction reads values from the database (tagged with prepare_seq) it makes use of the CommitCache to figure out if the commit_seq of the value is in its read snapshot.

The WritePrepared policy still can suffer from memory saturation since it keeps buffering the data until the prepare stage. Nevertheless, it helps transitioning from WriteCommitted to WriteUnprepared write policy.

A.1.2 WriteUnprepared

The idea behind the WriteUnprepared policy is to write the updates in the database at transactions' write stage, thereby bypassing the write queue in the commit phase. When a WriteUnprepared transaction's `WriteBatch` reaches a configurable threshold, the transaction writes the batch in the WAL and MemTable, respectively. This is similar to the WritePrepared transactions but instead of the prepare phase we emit updates in the write stage. These updates are also associated with a sequence number (*unprep_seq*) which is stored along with the updates in the WAL. Transactions keep all these unpreprared sequence numbers associated with its updates in order to track the

[1] *Commit marker* is a record that indicates that a transaction has been successfully committed.

[2] The CommitCache is an optimized, lock-free, in-memory data structure that keeps the recent commit entries (sequences). Its role is to distinguishes the data that are committed and thus, serve snapshot requests and/or transactions' rollbacks. RocksDB's design relies on the assumption that old data in the database – that are not present in the CommitCache – are supposed to be committed.

list of its unprepared batches that are written to the database.

At prepare phase the remaining updates in the transaction's batch are appended to the WAL and also a *prepare marker* is also appended to indicate that the transaction is now prepared. This is necessary because after a crash RocksDB may be configured to recover all successfully prepared transactions. This marker distinguishes the prepared from the unprepared transactions in case of a crash; the unprepared transactions in the WAL will be simply discarded during the recovery while, upon configuration, the prepared transactions can return to the application so that the application can perform the correct action.

During commit, the CommitCache needs to be updated as previously. The difference is that a transaction can potentially have multiple unprepared sequence numbers associated with it. So, in the CommitCache all (*unprep_seq*, *commit_seq*) pairs are added.

A.2 Logging Protocol: Synchronization of Concurrent Writers

RocksDB is an embeddable database that supports parallel transactions. When multiple clients (*write threads*) are active, RocksDB implements a leader election-based protocol to support atomic and serial logging to the WAL. This protocol implements *group commits* only in Pessimistic Concurrency Control by merging multiple transactions' updates into a single object which, thereafter, atomically is appended to the WAL.

A high level overview of this protocol in a multithreaded environment is shown in Figure A.1. In our example we have multiple clients operating in the database in parallel that enter their commit phase (dashed green lines). All write threads will eventually join the batch group and only one of them (typically the first thread that will enter the group) will be designated as the *leader*. The remaining threads, and any other threads that may also join the batch group later, will block (*awaiting state*) until they receive some notification from the current elected leader.

At this point of the execution, the leader is the only active thread and it is responsible for forming a *write group*. A write group is a logical group of threads that consists of the leader and/or a set of other – currently blocked threads – the *followers*. The selection of the followers is done by the leader and typically goes from old to new. The

leader will now perform the group commit by taking all `WriteBatches` of the write group, concatenating them together and writing this blob out to the WAL. The protocol allows the group to grow up to a maximum size, but if the original leader's batch is small, the protocol limits the growth in order not to slowdown small writes too much. Therefore, there is no guarantee that the group will contain all or some of the current blocking threads or only consist of the leader itself.

After finishing the write to the WAL, the leader may insert all merged updates into the MemTable or may notify the followers to insert their individual batches concurrently to the MemTable. That depends on the size of the write group and the MemTable's capability to support parallel writes. In default settings, concurrent writers to MemTable are the norm. The writes to the WAL and the writes to the MemTable should always be pipelined.

After successfully updating the WAL and the MemTable, the leader unlinks from the write group all the waiting threads. If the threads are in a waiting state, they are now notified that their state is complete, that is all of their updates have been pushed to the database by the leader. Before leader exits the write group, it also elects the leader of the next write group if there are pending writers. The new elected leader in turn will form a new write group and repeat the same process.

The group commit helps issuing one I/O operation while logging multiple client's batches. Native RocksDB does not support parallel logging since all updates of a write group are appended to a single log file and only a write group can be active a time. Therefore, all I/Os are sequential. To increase throughput RocksDB provides *pipelined* writes to the WAL which are presented to more efficient in cases with concurrent writers. Particularly, once the previous writer finishes its WAL write, the next writer waiting in the write queue can start writing to the WAL while the previous write group still has its MemTable write ongoing. Using this configuration, there is an chronological overlap between the current writer's logging operation and the previous writer's insertion to the MemTable. However, even in this configuration, I/Os to the WAL are still sequential and each leader will merge its write group's updates.

Lastly, RocksDB recently introduced *concurrent* writes to the WAL by keeping two write queues instead of one (which is the default). However, this configuration requires that one write queue handles only updates that bypass MemTable (e.g. writes at transaction's prepare phase). Additionally, each write queue will still compete to acquire an exclusive write lock on the current log file therefore an actual "parallel" write to the WAL is not implemented.

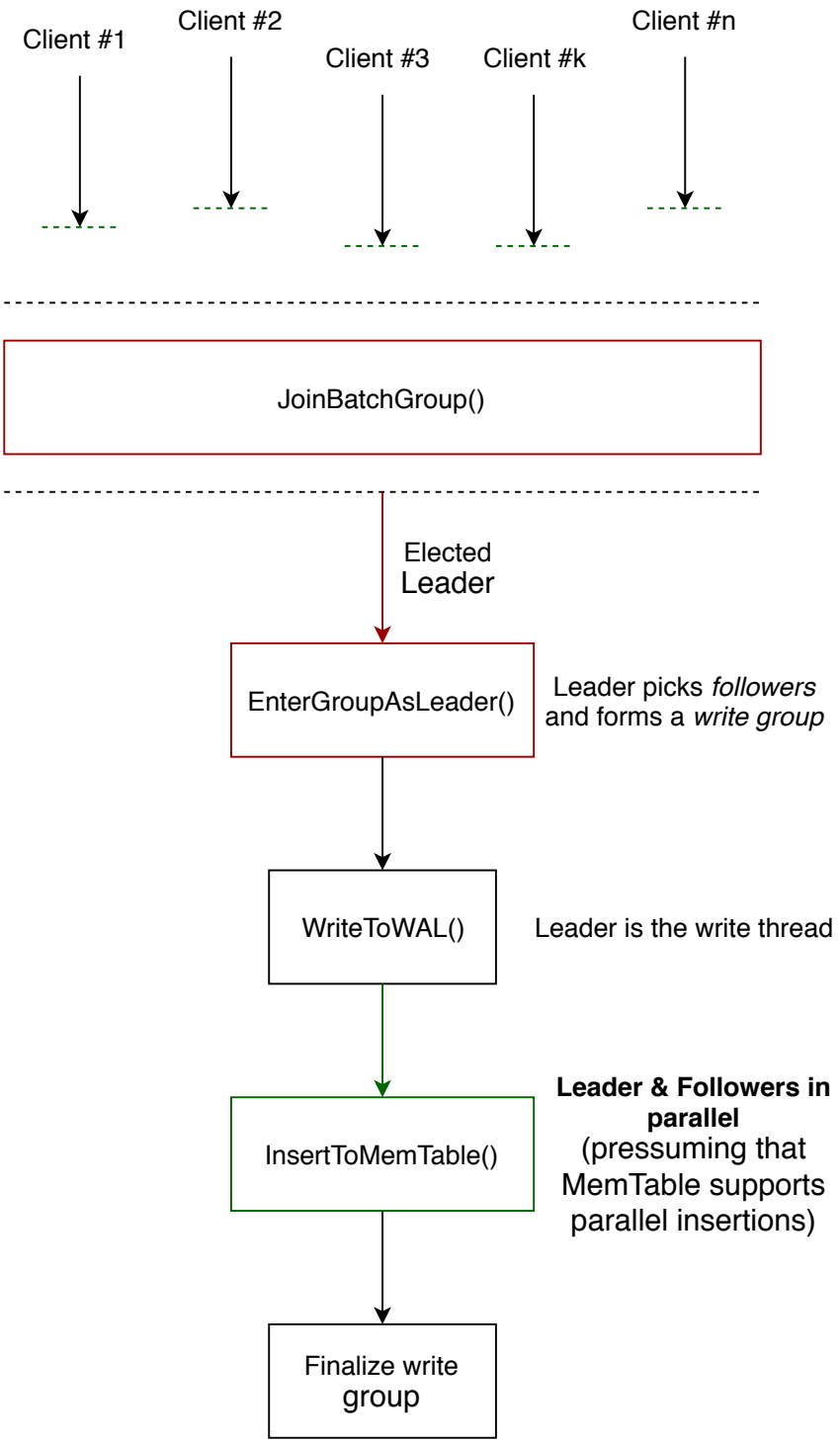

Figure A.1: RocksDB Logging Protocol

A very interesting detail we found out during this project is that the group commit is disabled when using optimistic transactions. The reason is that the conflict checking only checks for conflicts searching the MemTable and will not search for conflicts among the write group. Given that OCC does not take locks, it would be possible that

the aforementioned protocol would allow write threads with conflicting operations to exist in the same write group. Consequently, given that MemTable can be updated in parallel but the WAL is append only, would be possible a scenario where the database has partial interleaved updates of conflicting transactions. That would lead to an inconsistent and faulty state.

Appendix B

Intel SGX

B.1 Overview

Intel Software Guard Extensions (SGX) is a set of extensions in the instruction set of Intel's processors that offers integrity and confidentiality guarantees to a running software by keeping it isolated, hence protected against privileged – potentially malicious – code (BIOS, drivers, OS, hypervisors, System Management Mode, Intel Management Engine, etc.) and, by extension, against any remote attack.

The central idea of SGX is the hardware's establishment of a protected environment, called *enclave*, that contains any code and data of which we want to preserve their security guarantees. The main difference with its predecessors (TPM [15] and TXM [23]) is the amount of code covered by the attestation which is in the *Trusted Computing Base* (*TCB*) for the system using hardware protection. Specifically, TPM covers all running software while TXT covers the code inside a virtual machine. On the other hand, SGX uses enclaves.

In brief, the software to be executed with confidentiality and integrity guarantees is loaded from the untrusted system software. Initially, the CPU is asked to copy the data from the unprotected memory to the enclave memory and associates this memory with the enclave. After copying all data to enclave memory has been completed, the enclave is marked as initialized. At the enclave's loading phase the CPU cryptographically hashes the enclave's contents. The derived hash is the *measurement hash* can be used by a third party to perform software attestation in order to establish trust in an enclave.

In addition, enclaves also support sealing which allows an enclave to securely persist data (secrets) on the local host. Sealed data is confidentiality-and integrity-protected, but not rollback-protected.

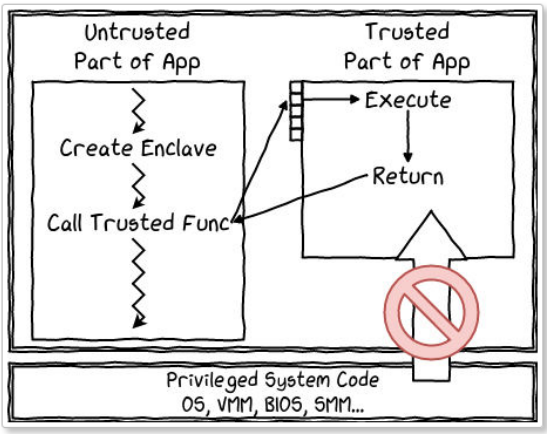

Figure B.1: Execution with SGX (source [10]).

Figure B.2 illustrates the SGX Memory Layout. SGX-enabled processors have a Processor Reserved memory (PRM); a continuous range of (subset) DRAM that is accessed exclusively by the enclave's running code. PRM is used to store enclaves' code and data and access from the OS, the SMM code and DMA are restricted.

Each enclave needs to place its running code and associated data inside the PRM. A subset of PRM, Enclave Page Cache (EPC), is dedicated to keep enclaves' data. EPC is splitted into *pages*[1] It is worth noting that EPC is managed by the system software that invokes SGX instructions to allocate pages to enclaves. The EPC pages of an enclave are not visible to non-enclave software or even other enclaves. In general non-enclave software has no access to the EPC.

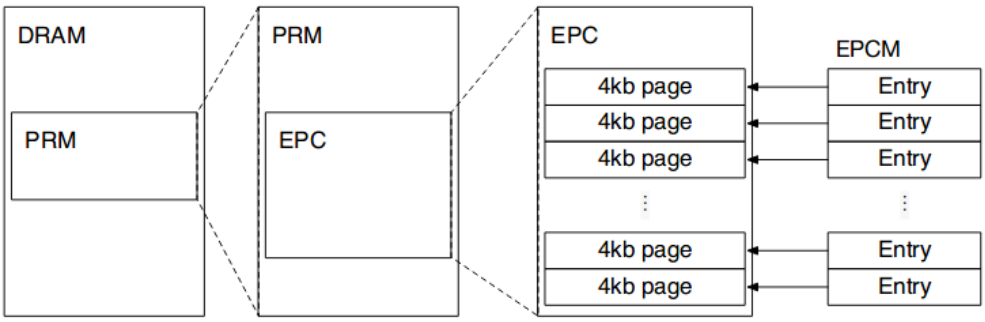

Figure B.2: SGX Memory Layout (source [20]).

Except for the EPC, SGX records information about the system software's allocation decisions for each EPC page in the Enclave Page Cache Map (EPCM). The EPCM is an array with one entry per EPX page as shown in Figure B.2. Normally, EPCM is invisible to any software and its purpose is to enable SGX to perform security checks,

[1] Similar to the pages introduced by the classic memory management paradigm [44].

e.g. ownership of a page (isolation guarantees), or an attempt to allocate a page that is already allocated by an enclave. Lastly, SGX keeps meta-data for each enclave into the SGX Enclave Control Structure (SECS). Enclaves cannot access these pages but they are really important to ensure all the aforementioned security attributes of SGX.

Appendix C

Additional Evaluation Measurements

C.1 Tables with Aborted Transactions under Optimistic Concurrency Control

Below we present the tables with the percentages of the aborted transactions while conducting the experiments under the Optimistic Concurrency Control.

10 `PUT`s per Transaction			
Threads	Conflicts	OCC Aborted Txns (%)	Native Aborted Txns (%)
1	5%	0%	0%
	10%	0%	0%
2	5%	0.6%	∼0%
	10%	1.29%	∼0%
4	5%	0.95%	∼0%
	10%	1.81%	∼0.1%
8	5%	1.06%	∼0%
	10%	2.02%	∼0.1%

100 `PUT`s per Transaction			
Threads	Conflicts	OCC Aborted Txns (%)	Native Aborted Txns (%)
1	5%	0%	0%
	10%	0%	0%
2	5%	0.8%	0%
	10%	1.61%	∼0%
4	5%	1%	∼0.1%
	10%	1.55%	∼0.1%
8	5%	1.7%	∼0.1%
	10%	4%	∼0.1%

500 `PUT`s per Transaction			
Threads	Conflicts	OCC Aborted Txns (%)	Native Aborted Txns (%)
4	10%	6.3%	6.7%
8	10%	12.8%	14.2%

1000 `PUT`s per Transaction			
Threads	Conflicts	OCC Aborted Txns (%)	Native Aborted Txns (%)
4	10%	11%	11.7%
8	10%	23%	23%

2000 `PUT`s per Transaction			
Threads	Conflicts	OCC Aborted Txns (%)	Native Aborted Txns (%)
4	10%	31.5%	31.1%
8	5%	26.9%	24.3%