Tiago Marques do Vale

Master of Science

Executing requests concurrently in state machine replication

Dissertação para obtenção do Grau de Doutor em

Informática

Orientador: João Manuel dos Santos Lourenço,
Professor Auxiliar, Universidade Nova de Lisboa

Co-orientador: Ricardo Jorge Freire Dias,
Engenheiro de Software Sénior, Suse Linux

Júri

Presidente: Luís Caires, Universidade Nova de Lisboa
Arguentes: Aleksandar Dragojevic, Microsoft Research
Alysson Neves Bessani, Universidade de Lisboa
Vogais: João Manuel dos Santos Lourenço, Universidade Nova de Lisboa
Luís Eduardo Teixeira Rodrigues, Universidade de Lisboa
Nuno Manuel Ribeiro Preguiça, Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Maio, 2019

Executing requests concurrently in state machine replication

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

State machine replication is one of the most popular ways to achieve fault tolerance. In a nutshell, the state machine replication approach maintains multiple replicas that both store a copy of the system's data and execute operations on that data. When requests to execute operations arrive, an "agree-execute" protocol keeps replicas synchronized: they first agree on an order to execute the incoming operations, and then execute the operations one at a time in the agreed upon order, so that every replica reaches the same final state.

Multi-core processors are the norm, but taking advantage of the available processor cores to execute operations simultaneously is at odds with the "agree-execute" protocol: simultaneous execution is inherently unpredictable, so in the end replicas may arrive at different final states and the system becomes inconsistent. On one hand, we want to take advantage of the available processor cores to execute operations simultaneously and improve performance. But on the other hand, replicas must abide by the operation order that they agreed upon for the system to remain consistent. This dissertation proposes a solution to this dilemma. At a high level, we propose to use speculative execution techniques to execute operations simultaneously while nonetheless ensuring that their execution is equivalent to having executed the operations sequentially in the order the replicas agreed upon. To achieve this, we: (1) propose to execute operations as serializable transactions, and (2) develop a new concurrency control protocol that ensures that the concurrent execution of a set of transactions respects the serialization order the replicas agreed upon. Since speculation is only effective if it is successful, we also (3) propose a modification to the typical API to declare transactions, which allows transactions to execute their logic over an abstract replica state, resulting in fewer conflicts between transactions and thus improving the effectiveness of the speculative executions.

An experimental evaluation shows that the contributions in this dissertation can improve the performance of a state-machine-replicated server up to ≈ 4×, reaching up to ≈ 75% the performance of a concurrent fault-prone server.

Keywords: Replication, concurrency, state machine replication, transactional model

Resumo

A replicação de máquina de estados é uma das maneiras mais populares de garantir tolerância a faltas. Em suma, a abordagem consiste em manter várias réplicas que armazenam uma cópia dos dados do sistema e executam operações sobre esses dados. Quando chegam pedidos para executar operações, um protocolo "concorda-e-executa" mantém as réplicas sincronizadas: primeiro as réplicas acordam uma ordem na qual vão executar as operações, e depois executam as operações uma de cada vez na ordem acordada, de modo a garantir que todas as réplicas produzem o mesmo estado final.

Processadores com vários núcleos são a norma, mas tirar partido deles para executar operações simultaneamente vai contra o protocolo "concorda-e-executa:" execução simultânea é inerentemente imprevisível, portanto as réplicas podem produzir estados finais diferentes e o sistema fica inconsistente. Por um lado, queremos tirar partido dos vários núcleos para executar operações simultaneamente e melhorar o desempenho. Por outro, as réplicas têm de respeitar a ordem de operações que acordaram para o sistema se manter consistente. Esta dissertação propõe uma solução para este dilema: utilizar técnicas de execução especulativa para executar operações simultaneamente mas garantindo que essa execução é equivalente a ter executado as operações sequencialmente na ordem acordada pelas réplicas. Para atingir este fim (1) propomos executar operações como transações serializáveis, e (2) desenhamos um novo protocolo de controlo de concorrência que garante que a execução concorrente das transações respeita a ordem acordada pelas réplicas. Como a execução especulativa só é eficaz se for bem sucedida, também (3) propomos uma modificação à interface típica para declarar transações, que permite as transações executarem a sua lógica sobre um estado abstrato, o que resulta em menos conflitos entre transações e portanto numa melhoria da eficácia da execução especulativa.

Uma avaliação experimental mostra que as contribuções nesta dissertação podem permitir melhorar até $\approx 4\times$ o desempenho de um servidor tolerante a falhas, chegando até $\approx 75\%$ do desempenho de um servidor concorrente mas intolerante a faltas.

Palavras-chave: Replicação, concorrência, replicação de máquina de estados, modelo transacional

Contents

List of Figures

Acronyms

2PL	Two-phase Locking
HTM	Hardware Transactional Memory
LSD	Lazy State Determination
OCC	Optimistic Concurrency Control
PCC	Pot Concurrency Control
Pot	Preordered transactions
ROT	Rollback-only Transaction
STM	Software Transactional Memory
TM	Transactional Memory

*

Chapter 1

Introduction

1.1 Context

We, the human society, increasingly depend on computing systems and the Internet to support many aspects of our existence. Computers power virtually everything from entertainment (e.g. Netflix [52]) to social (e.g. Facebook, Twitter [26, 66]) and economic interactions (e.g. Amazon, eBay [4, 23]). As a consequence, the systems we build operate on a scale like never before.

The ubiquity of computer systems, and the scale at which they operate, mean that failures are common and inevitable. Machines may fail due to a variety of reasons, such as hardware malfunction, or software bugs. The following is a quote from Jeff Dean, a Google Senior Fellow as of March 2017:

> In each [Google] cluster's first year, it's typical that 1,000 individual machine failures will occur. *You have to provide reliability on a software level.* [17, 60]
>
> — Jeff Dean, Google Senior Fellow.

At first glance it might seem that the majority of systems should not experience failures as commonly because they do not serve a Google-scale user base. But with the advent of Infrastructure as a Service (IaaS) products such as Amazon EC2 [3] and Google Compute Engine, [29] small and medium-sized companies are deploying their systems in IaaS offerings. [6] These companies are also effectively operating their systems on large-scale clusters where failures are common. [28]

Since we seem to rely more and more on computer systems, they should tolerate failures, i.e. operate correctly even in their presence. Redundancy is the typical approach to achieve fault tolerance. For example, if we do not want to lose our favorite video file (e.g. a cat playing a piano),[1] we should keep multiple copies of the video file. In the same vein, to tolerate the failure of a computation it should be performed by multiple entities. The context of this dissertation and the research it describes is fault tolerance of computation.

1.2 Problem statement

State machine/active replication [41, 58] is the standard way to build fault-tolerant systems. In a nutshell, the system maintains multiple replicas that both store a copy of the system's data and execute operations on that data. When requests to execute operations arrive, an "agree-execute" protocol keeps replicas synchronized: they first agree on an order to execute the incoming operations, and then execute the operations in the agreed upon order, so that every replica reaches the same final state. As a result, the replicas execute operations sequentially. However, replicas are made up of multi-core processors, since the speed of an individual processor core has plateaued. But taking advantage of the available processor cores to execute operations simultaneously is at odds with the "agree-execute" protocol: simultaneous execution is inherently unpredictable, so in the end replicas may arrive at different final states and the system becomes inconsistent. So we have a dilemma. On one hand, we want to take advantage of the available processor cores to execute operations simultaneously. But on the other hand, replicas must abide by the operation order that they agreed upon for the system to remain consistent.

1.3 Thesis statement and contributions

The research in this dissertation proposes a solution to the dilemma of exploiting multi-core processors in state machine replication. As such, this dissertation claims the following thesis to be true:

> *It is possible to improve the performance of the state machine replication's execution phase by taking advantage of multiple processors to execute operations concurrently, while at the same time maintaining a sequential programming model and respecting the safety properties of the state machine replication approach.*

To back the thesis statement, we propose to use speculative execution techniques to execute operations simultaneously while nonetheless ensuring that their execution is equivalent to having executed the operations sequentially in the order the replicas agreed upon. To achieve this, the research in this dissertation: (1) proposes to execute operations as serializable transactions, and (2) develops a new concurrency control protocol, Preordered transactions (Pot), that ensures that the concurrent execution of a set

[1] Adorable!

of transactions respects a specific serialization order, which in this case is the order replicas agreed upon. Furthermore, since speculation is only effective if it is successful, the research in this dissertation also (3) proposes a modification to the typical API to declare transactions (e.g. begin, read, write, and commit). The proposed API, called Lazy State Determination (LSD), allows transactions to execute their logic over an abstract replica state, which results in fewer conflicts between transactions and consequently in more successful speculative executions.

An experimental evaluation of a prototype implementation of Pot and LSD (Section 5.3) shows that they can improve the performance of a fault-tolerant server up to $\approx 4\times$. From other point of view, the experimental results show that a fault-tolerant server can go from being $\approx 8\times$ slower (using sequential execution) to only $\approx 25\%$ slower (using concurrent execution with the contributions in this dissertation) than a concurrent fault-prone server.

In summary, the research described in the following chapters of this dissertation makes the following contributions:[2]

- Pot, described in chapter 3, a methodology and associated concurrency control protocol that ensures that the concurrent execution of requests in the state machine replication model is safe. This contribution has been published in peer-reviewed venues; [67, 68] and

- LSD, described in chapter 4, an extended transactional API and associated concurrency control protocol that enhances the effectiveness of Pot. This contribution has been submitted to peer-reviewed venues.

The contributions described in this dissertation have contributed to the genesis of the "HiPSTr: High-Performance Serializable Transactions" national research project, funded by Fundação para a Ciência e a Tecnologia under contract PTDC/CCI-COM/32456/2017 / LISBOA-01-0145-FEDER-032456. Part of the work described in this dissertation has been conducted in the context of the HiPSTr research project.

This research was partially funded by Fundação para a Ciência e Tecnologia under the Ph.D. scholarship SFRH/BD/84497/2012, and the POCI-COMPETE2020 projects UID/CEC/04516/2013 and PTDC/CCI-COM/32456/2017 / LISBOA-01-0145-FEDER-032456.

1.4 Outline

The rest of this document is structured as follows. Chapter 2 succinctly describes the context of this dissertation's contributions. Chapter 3 presents our first proposal, Pot, which

[2] We assure you that, despite the names of the contributions, the author of this dissertation only consumes legal drugs, such as alcohol and caffeine, for recreational purposes. Well, to be entirely honest, caffeine was consumed whilst producing this document. Does that count as recreation?

uses speculative techniques to ensure that the concurrent execution of a set of transactions respects a specific serialization order. Chapter 4 presents our second proposal, LSD, which improves the efficacy of Pot's speculation. Chapter 5 realizes the full vision of this dissertation's research by combining Pot and LSD. Chapter 6 discusses related work, and chapter 7 concludes the dissertation.

Chapter 2

Preliminaries

This chapter describes the context of the contributions described in this dissertation. Section 2.1 presents the state machine replication model. Section 2.2 overviews the transactional model.

2.1 State machine replication

As society increases its reliance on computer systems, these should strive to be as fault tolerant as they can. State machine replication [41, 58] is a general approach to construct fault-tolerant systems. To understand how it works, let us go through a thought experiment.

Consider a computer system that provides some service, e.g. a bank account. The service maintains state, such as the account's balance, and provides operations that manipulate that state, such as an operation to withdraw money. A simple realization of this system is to have a client-server architecture. The server consists of a single machine that stores the service's state and execute the service's operations. The clients send requests to the server machine, which executes the requested operations and replies with their results.

However, now consider that some accident happens and as a consequence we lose the server machine's storage. We lost our bank account and it is unacceptable. To tolerate situations like these, we need to have redundant copies of the service's state stored elsewhere. We can add an additional machine to our system to stores a redundant copy of the service's state.

This approach is called primary-backup. [2] The idea is that one of the machines will execute the service's operations on behalf of the clients—the primary—while one or more additional machines store redundant copies of the state—the backups. The primary ships the updates it performs to the state to the backups in order to keep their redundant copy of the state up to date. Should the primary be lost, the system can tolerate its fault by promoting one of the backups to primary.

Now consider a more subtle scenario where the primary continues operating but performs its jobs incorrectly, e.g. due to some bug in its implementation, or some transient error such as a bit flip. Under this circumstance, it is possible for the primary to incorrectly perform an operation's computation. For example, it can update the bank account's balance to an incorrect value. As soon as the incorrect computation reaches the backups, every copy of the service's state is now wrong. The primary-backup scheme we described is able to tolerate the fault of the service's data as long as there we have one more backup. However, it is not able to tolerate the fault of the primary's *computation*.

The state machine replication approach can tolerate faults of both the service's data and computation. The general idea is to take a system and replicate both its data and operations across a set of replicas. Since these replicas each store a copy of the system's data, this data is not lost as long as there is at least one replica that does not fail. And since replicas also execute every operation submitted to the system, the system can continue operating as long as a majority of correct replicas execute the operations.

In the primary-backup approach, executing the clients requests was the primary's responsibility. The primary would update the various redundant copies of the service's state so they remained synchronized with the primary copy. In state machine replication, executing the clients requests is every replica's responsibility. A question arises naturally: how are the various copies of the state kept synchronized? Clients cannot simply send their requests to all replicas in an uncoordinated manner. The replicas may receive, and process, the clients requests in different orders. This can lead the replicas' state to diverge.

For example, consider a bank account with a balance of 100€. Imagine there are three clients requesting three different operations, and that the system has three replicas. The first client, D, wants to deposit 100€. The second client, W, wants to withdraw 150€. The third and final client, C wants to credit the account with an interest rate of 10%. One replica may execute the request of D, (balance is now 200€) then W, (balance is now 50€) and finally C, leading to a final balance of 55€. Other replicas may execute the requests in different orders, e.g. (D, C, W) leads to a balance of 70€, whereas (C, D, W) leads to a balance of 60€. This could lead to a situation where each replicas has a different state!

The state machine replication approach deals with this issue by having the replicas first reach a consensus on which order to execute incoming requests. Once the replicas have agreed on a common order, they execute the requests in that order, ensuring that every correct replica reaches the same final state. Figure 2.1 depicts an overview of the state machine replication approach. A set of clients (left) submit requests to the system concurrently. The replicas (right) run an agreement protocol, e.g., Paxos, [42] that totally orders incoming requests. The replicas execute the requests sequentially in the agreed upon order, ensuring that each correct replica arrives at the same final state. Essentially, we can divide state machine replication in two phases: first, the agreement phase, where

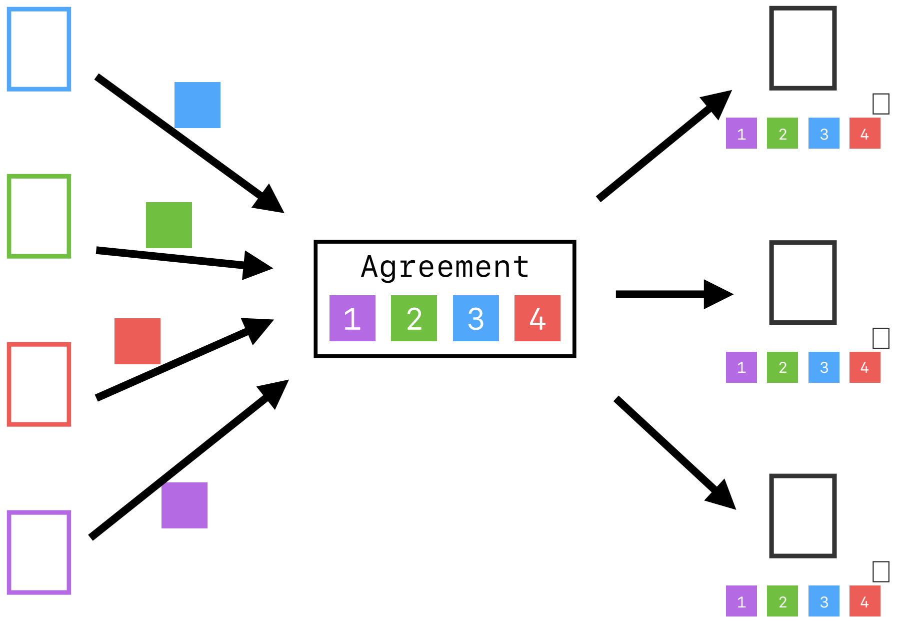


Figure 2.1: Overview of the state machine replication approach. Clients (left) submit requests to the system concurrently. An agreement protocol orders the concurrent requests. The replicas (right) execute the requested operations in the agreed upon order, which keeps their individual state synchronized.

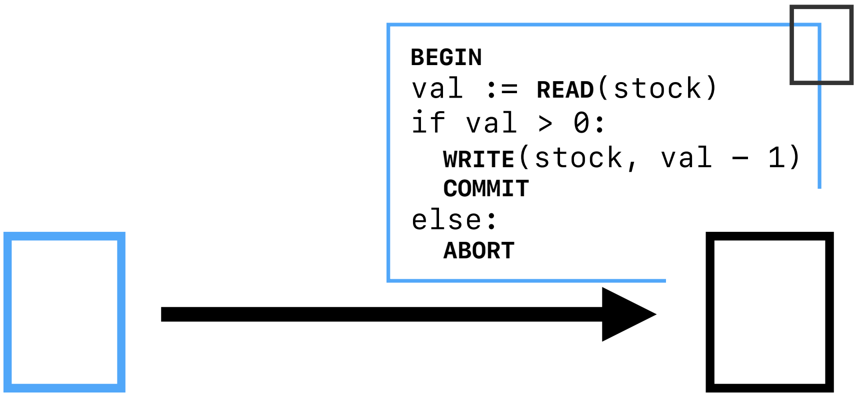
replicas agree on an order for all requests; followed by the execution phase, where replicas execute the requested operations in the order agreed upon in the previous phase.

The concern of this dissertation's research is state machine replication's execution phase. Specifically, the tension between the fact that the replicas have multi-core processors and the requirement that replicas execute operations sequentially. The research in this dissertation proposes a way to exploit the replicas' multi-core processors to execute operations concurrently such that the outcome is still equivalent to having executed those operations sequentially. Even though state machine replication allows us to build systems that tolerate from simple crashes to more complex failures [43] such as data corruption and software errors, this dissertation, and its research, is orthogonal to any particular type of fault.

2.2 Transactions

The research in this dissertation proposes a way to safely execute operations concurrently under the state machine replication model. A central part of that way is to execute operations as serializable transactions, [24, 55] henceforth referred to simply as transactions.

A serializable transaction is a sequence of actions that appear to execute instantaneously as a single, indivisible, operation—the system itself will actively ensure this illusion. For example, a transaction that implements the action of withdrawing money from a bank account will first check if there is enough money in the account, and if so,



```
BEGIN
val := READ(stock)
if val > 0:
  WRITE(stock, val - 1)
  COMMIT
else:
  ABORT
```

Figure 2.2: A typical transactional API consisting of the begin, read, write, commit, and abort operations. The example logic is a simplified transaction that buys one item from an e-commerce application.

deduct the withdrawn amount from the balance. As per the definition of a serializable transaction, the act of checking if the account has enough money and the act of deducting the withdrawn amount appear to happen simultaneously. This means that the transaction will not experience any interference from other transactions, such as a modification of the account's balance between checking whether there is enough money, and updating the account's balance.

A developer specifies a transaction's logic using a well defined API. Without loss of generality, the API is typically composed of five operations. The begin operation starts a transaction. A transaction can observe and modify the state using the read and write operations, respectively. A transaction finishes the execution of its logic with the commit operation, which atomically applies all the transaction's modifications to the state. Alternatively, a transaction may finish with the abort operation, which reverts all the modifications the transaction performed.

Consider the example in Figure 2.2, which shows the API in action to define a simplified transaction that buys one item from an e-commerce application. The transaction checks whether there is enough stock available, and if there is, the transaction decrements the amount in stock.

Given that a transaction is an indivisible unit of work, we can reason about the system as if it executes transactions one at a time. However, a system that actually executes transactions sequentially is inefficient. In practice, systems execute transactions concurrently, as it allows the system to make better use of the available resources. For example, if a transaction stalls waiting for disk access, the system can keep the processor performing useful work by executing another transaction while the aforementioned transaction waits. The system can also execute transactions in parallel to take advantage of the fact that modern processors have multiple cores.

But if transactions execute concurrently, how is it that they appear to execute one a time? For instance, in the example in Figure 2.2 we do not see any measures being taken to ensure the serializable semantics. Indeed, the developer only needs to specify *what* should be atomic—the transactions—and is spared from specifying *how* to achieve the desired atomicity. The system will ensure serializability automatically under the hood. This approach greatly simplifies the developer's job, and stands in stark contrast to the typical approach of using explicit locking.

To transparently achieve serializable semantics, the system runs a concurrency control protocol [11] that kicks in during the transactional API calls. The job of the concurrency control protocol is to ensure that transactions appear to execute one at a time. A central part of achieving this illusion is making sure that a transaction that commits successfully did not experience any interference from other concurrent transactions. In the example in Figure 2.2, this means that the stock value should not change from the moment the transaction observes it $\big(\mathsf{read}(stock)\big)$ until the transaction commits. There are various concurrency control protocols, but one can group them broadly in two families: pessimistic or optimistic.

Pessimistic protocols follow a cautious route, and take measures to ensure that interference does not happen. In our example, an example of a pessimistic approach is for the transaction to acquire a read lock when it observes the stock value, preventing any other transaction from modifying the stock value until the transaction commits—at which point it releases the lock. [24] The optimistic family of protocols take the opposite approach: they assume interference will be rare, and thus allow transactions to execute without synchronization until they attempt to commit—in other words, transactions execute speculatively. When a transaction attempts to commit, the system atomically verifies if the transaction's speculative execution is still valid, and if so, commits the transaction. If not, the system aborts the transaction, which can then be retried. In our example, when the transaction issues the commit operation, the system verifies whether the stock value that the transaction observed during its speculative execution remains unchanged before allowing the transaction to commit successfully. If the stock value remains unchanged, the system commits the transaction. Otherwise, the system aborts the transaction.

The system's concurrency control protocol, regardless of whether it is pessimistic or optimistic, ensures that transactions are serializable. This means that transactions appear to execute one a time in some order. Any order is deemed correct. The idea in this dissertation's research is to execute operations concurrently as transactions under the state machine replication model. However, if the system only ensures that transactions appear to execute one a time in *some* order, it is possible for the replicas to execute transactions in *different* orders. However, we have already established (in the previous Section) that each replica must execute the requests (transactions) in the same order to respect the state machine replication approach. In this dissertation's research we rework the concurrency control protocol to ensure that transactions, despite executing concurrently, respect a particular serial order: the order that the replicas agreed upon.

In the research we describe in this dissertation, we also rethink the transactional API.

The system's perception of what the transaction does is limited by the information that passes through the API calls, which can lead the system to conservatively prevent concurrency. Take the example in Figure 2.2 again, assuming that there is enough stock. A fair description of its transaction is that it "decrements the amount in stock if there is enough stock available." However, during the transaction's execution what the system sees is that the transaction observes a specific stock value and subsequently modifies the stock. The system is unaware, for instance, that the new stock value is a function of the previous value, nor that the transaction is only really interested in checking whether there is enough stock. As a consequence, the system prevents the concurrent execution of transactions that attempt to buy the same item, regardless of whether there is enough stock to allow all the transactions to concurrently buy the item. For example, if a transaction observes the stock value as five, it computes the new stock value to be four. The new value, four, only makes sense in a state where the current value is five, so for this transaction to commit other transactions must not modify the stock value.

Our research enhances the transactional API to allow the application to provide more information to the system, so that it can reap the available concurrency as much as possible. For example, the enhanced API allows the transaction to convey to the system that it only requires that there is enough stock, i.e. it is greater than zero, and to convey to the system how the new stock value is computed based on the previous stock value. This allows the system to successfully commit transactions that concurrently buy the same item, as long as there is enough stock.

Chapter 3

Pot: Preordered transactions

In this chapter we present Preordered transactions (Pot), a methodology to achieve deterministic concurrent execution of transactions. We describe how to apply Pot's methodology to optimistic concurrency control using two key techniques: *ordered commits* and *transaction modes*. Finally, we evaluate our implementations of two Pot TM prototypes using STAMP [50], STMBench7 [31], and microbenchmarks.

The chapter is organized as follows. We start by introducing and motivating the need for Pot in Section 3.1. Section 3.2 presents Pot's design, namely its sequencer (Section 3.2.1) and concurrency control protocol (Section 3.2.2). Section 3.3 highlights the challenges and details our implementation of Pot in an STM and an off-the-shelf HTM system. We conclude in Section 3.4, which reports an experimental evaluation of Pot.

3.1 Introduction

Serializable transactions [24, 55] are a viable mechanism to synchronize concurrent accesses to shared state due to an interesting trade-off between ease of use and performance. Programmers specify which portions of code should be atomic (transactions) *without* worrying how to enforce such atomicity. A concurrency control protocol enforces atomicity at runtime, providing the illusion that transactions execute one at a time.

The transactional model has been widely used in databases, where transactions are the system's units of work. In the last decade, the transactional model has found its way to general-purpose programming in the form of Transactional Memory (TM) [35, 62]. TM is becoming mainstream, as processors from Intel and IBM already provide support for HTM [14, 72], the GCC has experimental support for TM (using either STM or HTM) [27], and there is ongoing work in integrating TM language constructs in C/C++ [12].

```
1 Thread 1                1 Thread 2
2 if test it then         2
3                         3 change it
4     assume it           4
```

a Atomicity violation.

```
1 Thread 1                1 Thread 2
2                         2 use it
3 initialize it           3
4                         4
```

b Order violation.

Figure 3.1: Example of the most common concurrency bugs [Lu et al. 2008], with transactions in *italic*. In (a) the assumption of a predicate is not atomic with its test. In (b) thread 2 uses a resource before thread 1 initializes it.

Although the transactional model provides a simple programming model, it inherits the nondeterministic behavior of concurrent execution. Specifically, the order in which transactions appear to execute depends on the nondeterministic interleavings of the threads that execute the transactions at runtime, so different executions of the same program with the same inputs can yield different outcomes. In this chapter we focus on building a transactional system that ensures that data race-free programs execute according to a deterministic transaction serialization order.[1]

Having a system that ensures a deterministic transaction serialization order has at least two benefits: (1) we can execute multiple replicas of a multithreaded application for fault tolerance [58], and (2) it helps debugging, or prevents, the most common concurrency bugs [46]. Executing multiple replicas for fault tolerance relies on the assumption that correct replicas always yield the same outputs. With a deterministic transaction serialization order this assumption is *not* broken under multithreaded execution, so replicas do *not* need to fall back to sequential execution to ensure correctness. Consequently, replicas potentially make better use of the available resources such as multicore processors. Regarding concurrency bugs, Figure 3.1 depicts the two most common concurrency bugs (amounting to 97% of the non-deadlock bugs) found in a study of 4 real-world applications [46], with transactions highlighted in *italic*. Figure 3.1a shows an example of an atomicity violation. Thread 1 tests some predicate, and then executes code that assumes that it is true. Thread 2 executes code that changes the predicate's outcome. If thread 2 interleaves thread 1 after the predicate test, but before the "then branch," thread 1 will execute code that assumes the predicate is true while it is not, which can result in unexpected behavior. Figure 3.1b shows an example of an order violation. Thread 1 initializes some resource that thread 2 uses, but at runtime thread 2 attempts to use the resource before thread 1 initializes it. These concurrency errors are sensitive to thread interleavings, and in the particular case of TM, only manifest themselves in particular transaction serialization orders. Since the transaction serialization order is nondeterministic, the errors are difficult to reproduce and debug. With a deterministic transaction serialization order, the aforementioned errors either manifest themselves in every execution, or not at all, greatly simplifying the developer's work.

[1] This property is known as weak determinism [53].

In this chapter we present Pot, a system that enables deterministic execution of transactions.

Pot uses the concept of *preordered transactions* as a principled approach to ensure a deterministic transaction serialization order. While traditional transactions provide the illusion of executing one at a time in *any* order, preordered transactions appear to execute in a specific, predefined, order.

To realize preordered transactions, Pot must address two key challenges: (1) guarantee that the predefined serial order is the same across executions, and (2) that the outcome of executing transactions is as if they executed serially in the predefined order. To ensure (1), Pot's *sequencer* assigns a sequence number to each new transaction. The sequence number reflects the transaction's place in a deterministic transaction serialization order. To ensure (2) efficiently, Pot executes transactions concurrently and relies on a new concurrency control protocol that guarantees that the outcome is equivalent to the order defined by the sequencer. Pot's concurrency control protocol relies on two key techniques: *ordered commits* and *transaction modes*. Ordered commits force transactions to commit according to the predefined serialization order. Transaction modes leverage the key insight that, at any given time, there is always one transaction that is "the next allowed to commit." Pot's concurrency control protocol executes that particular transaction as fast as possible, with virtually no concurrency control overhead (*fast mode*) while executing the other transactions using regular mechanisms to maintain correctness in the presence of the fast-mode transaction (*speculative mode*).

We built two Pot prototypes, one using STM and another using off-the-shelf HTM, and evaluate them with the popular STAMP benchmark suite [50] and STMBench7 [31]. Our Pot STM implementation clearly outperforms the state of the art in STM-based deterministic execution while simultaneously achieving deterministic execution with low overhead, providing promising evidence that using both STM and determinism to ease multithreaded programming may be practical. To the best of our knowledge, Pot also advances the state of the art by enabling deterministic execution of off-the-shelf HTM-based multithreaded programs for the first time.

3.2 Design

The standard Transactional Memory (TM) correctness criterion is opacity [30]. Broadly speaking, opacity is serializability [55] with the additional guarantee that transactions *never* observe inconsistent states, even if they would eventually abort. Traditional concurrency control protocols used to implement opaque transactions, such as Two-phase Locking [11] or Optimistic Concurrency Control [40], embrace opacity's flexibility and perform two tasks simultaneously while transactions are executing: (a) they compute the transaction serialization order (ordering), and (b) control the concurrent execution of transactions to respect that serialization order (concurrency control). Since ordering is intertwined with concurrency control, the final transaction serialization order depends on the nondeterministic interleavings that occur at runtime between transactions and thus varies from one execution to the next. We refer to this execution model as *traditional transactions*.

With *preordered transactions* the serialization order is independent of the interleavings that may occur between transactions because, unlike traditional transactions, preordered transactions already have a place in the serialization order before they are executed. Conceptually, preordered transactions have a two-phase execution model: (1) the *ordering phase* which defines every transactions' place in the serialization order, and (2) the *execution phase* where transactions execute concurrently in such a way that the outcome is equivalent to their sequential execution in the predefined order. Traditional concurrency control protocols *cannot* be used in the execution phase, because they implement both ordering and concurrency control. This chapter proposes a novel concurrency control protocol that can be used in the execution phase (Section 3.2.2).

3.2.1 Ordering phase: Pot sequencer

A consequence of decoupling ordering and concurrency control is that both the ordering and execution phase, where concurrency control occurs, can be performed separately by two different components. Ordering is performed by a *sequencer* component that computes some total order over the set of all transactions.

In the context of state machine replication, the sequencer maps directly to the agreement phase, which already totally orders transactions.

In the context of a general-purpose TM program, at first glance it seems that the sequencer needs to know which transactions will execute ahead of time, but we can devise generic sequencers that compute the transaction order on-the-fly by defining an order over the application threads and deriving the transaction order from it. For example, take threads t and u, with transactions $(a;b;c)$ and $(d;e;f)$ in their code, respectively. Consider a sequencer that orders threads using a round-robin scheme, i.e. $(t;u)$. This sequencer defines the transaction order $(a;d;b;e;c;f)$. Now consider that thread t only executes transaction c depending on some condition. The condition may be defined over global state, thread-private state, or a mixture of both. If the condition is over global state, the respective state must have been read within a transaction, e.g. transaction b, so the condition is always tested over the state resulting from the order $(a;d;b)$, yielding a deterministic result.[2] If thread t decides not to execute transaction c the order is $(a;d;b;e;f)$. If thread t's logic is "execute c or g" instead, the order is $(a;d;b;e;g;f)$.

The only requirement of a generic sequencer that derives the transaction order from the thread order is that the events of starting and stopping threads must be processed deterministically by the sequencer with respect to the transaction order. To do so, since transactions appear to execute in a deterministic order, Pot treats thread start/stop events

[2] Assuming the only source of nondeterminism is the transaction serialization order. Techniques to deal with other sources, e.g. randomness, are complementary to this work.

as if they are transactions. Take threads t and u, with transactions $(a;b;c)$ and $(d;e;f)$, respectively, where transaction b is the creation of a new thread v with transactions $(g;h)$. If we organize threads in a tree where the main thread is the root, the remaining threads are children of the thread that spawned them, and let the tree's post-order traversal specify the thread order, a round-robin sequencer defines the transaction order $(a;d;b;e;g;c;f;h)$.

It is also possible to use application-specific sequencers. For example, we may record the transaction commit order in a nondeterministic execution and then feed it to a sequencer to replay the recorded execution. We can also have sequencers that explicitly define a transaction order, e.g. $(a;b;c;d;e;f)$, but these need to take care because if a thread decides not to execute a transaction in the order then the program would hang waiting for it to execute. (We can detect this situation and abort the application with an error.)

This design for a TM sequencer works best for workloads in which threads perform transactions regularly. Optimizing for workloads with very heterogeneous thread behaviors is an open problem left for future work.

3.2.2 Execution phase: Pot Concurrency Control

Transactions may execute once they go through the ordering phase. At the core of the execution phase is a concurrency control protocol that guarantees equivalence to the serialization order defined in the ordering phase. The straightforward way to implement such concurrency control protocol is to simply execute transactions sequentially. However this approach is clearly suboptimal as it does not take advantage of the inherent parallelism present in today's multicore architectures.

This section describes Pot Concurrency Control (PCC), a new protocol that executes transactions concurrently while guaranteeing equivalence to the serial order defined by the sequencer. We design PCC by modifying Optimistic Concurrency Control (OCC), which works as follows. An OCC transaction consists of one, or more, speculative executions. A speculative execution is divided into three phases: (1) the read phase, (2) the validation phase, and (3) the write phase. The read phase records the objects read by the transaction in the transaction's read set. Write operations do not modify the shared state; instead the transaction defers its updates and logs them in its write set. Therefore locations that are both read and modified occur in both the read and the write sets. After the read phase, the transaction undergoes a validation phase where it checks whether any concurrently committed transaction's updates overlap with its read set. If so the transaction is aborted to respect opacity, and can be retried; otherwise it proceeds to the next phase. Finally, the transaction enters the write phase where it atomically updates all objects in its write set with the values buffered during the read phase.

We have chosen OCC as the base for PCC because OCC is suitable for dynamic transactions, i.e. transactions for which it is very difficult (or even impossible) to identify their

```
1  tx-begin($t$)
2
3  tx-write($t$, $o$, $v$)
4      deferred-write($o$, $v$, $W_t$)
5  tx-read($t$, $o$)
6      consistent-read($o$, $R_t$, $W_t$)
           or tx-abort
7  tx-commit($t$)
8      atomically
9          if validate($R_t$)
10             writeback($W_t$)
11         else
12             tx-abort
```

a OCC.

```
1  tx-begin($t$)
2      $sn_t \leftarrow sn$
3  tx-write($t$, $o$, $v$)
4      deferred-update($o$, $v$, $W_t$)
5  tx-read($t$, $o$)
6      consistent-read($o$, $R_t$, $W_t$)
           or tx-abort
7  tx-commit($t$)
8      wait until $sn_c$ = pred($sn_t$)
9      if validate($R_t$)
10         writeback($W_t$)
11         $sn_c \leftarrow sn_t$
12     else
13         tx-abort
```

b Speculative PCC.

```
1  when $sn_c$ = pred($sn_t$)
2      if validate($R_t$)
3          writeback($W_t$)
4      else
5          tx-abort
6  tx-write($t$, $o$, $v$)
7      direct-update($o$, $v$)
8  tx-read($t$, $o$)
9      direct-read($o$)
10 tx-commit($t$)
11     $sn_c \leftarrow sn_t$
```

c Fast PCC.

Figure 3.2: Methodology to transform Optimistic Concurrency Control (OCC) into Pot Concurrency Control (PCC). Figure 3.2a models a typical OCC transaction. sn_c represents the sequence number of the last committed transaction. sn_t, R_t and W_t represent the sequence number, read set, and write set of transaction t, respectively.

read/write sets in advance. Dynamic transactions are common in general-purpose TM-based programs due to aliasing and the unstructured nature of the heap. In fact, most STM and all existing HTM concurrency control protocols are optimistic.

Next, we present PCC incrementally. First, we describe the baseline OCC protocol in Section 3.2.2.1, and then present our methodology to transform the baseline OCC protocol into PCC by applying two key techniques: ordered commits, in Section 3.2.2.2, and transaction modes, in Section 3.2.2.3.

3.2.2.1 Baseline protocol

Consider the protocol depicted in Figure 3.2a, modeling a typical OCC scheme [20, 40]. The read phase occurs after *txn_start* and before either *txn_commit* or *txn_abort*, and consists of invocations to *txn_read* and/or *txn_write*. Both the validation and write phase occur during *txn_commit*.

Read phase. Write operations intending to update object o's value to v, buffer the update in W_t (Figure 3.2a, line 4). Read operations on an object o log the access in the transaction's read set R_t and return (a) the buffered value for o in the write set W_t, if existing, or (b) read a value of o from the shared state consistent with the rest of the read set (line 6). If it is not possible to read a consistent value the transaction aborts. For example, take two objects x and y, both initially 0. Transaction t observes $x = 0$. Meanwhile, another transaction commits and sets both x and y to 1. If transaction t attempts to read y it can either return 0 or abort, but it must never return 1, because $x = 0$ and $y = 1$ is not possible under opacity.

Validation phase. The validation phase iterates the read set and checks that the observed values are still coherent, i.e., all the observed values remain the same (line 9).
Write phase. If validation is successful then transaction t enters its write phase and directly updates the objects in its write set with the values buffered during the read phase, creating a new version of the shared state (line 10).
Correctness. This protocol guarantees opacity mainly due to the atomicity of the validation and write phases (lines 8–12). If the validation phase is successful then none of the read objects have been modified since the transaction's read phase. This means that the read phase happens in the same logical instant of the validation phase. Since the validation and write phase occur atomically, the write phase also happens in the same logical instant of the read phase. Therefore, transaction t appears to have been the sole transaction executing. Hence t is serialized after all the transactions that wrote the values t observed, and before any transactions that eventually observe the values t wrote.

3.2.2.2 Ordered commits

The OCC protocol described in the previous section provides the illusion that transactions execute one at a time. However, the order in which transactions appear to execute is not deterministic because it depends on the interleavings between transactions' operations that will occur at runtime.

To adhere to the serial order predefined in the ordering phase, we make two key observations: (a) OCC transactions only modify shared state during their write phase, and (b) each transactions' place in the serialization order depends on the relative order in which each transaction (atomically) performs its validation and write phase. If we restrict transactions to execute their validation and write phases in the order defined by the sequencer, we guarantee that the outcome is equivalent to the respective ordered sequential execution.

To transform the OCC protocol described in the previous section into PCC, we start by updating the *txn_start* operation to have an additional parameter, a sequence number sn, that reflects the order of transaction t in the serialization order defined by the sequencer (Figure 3.2b, line 2). Transaction t is preordered after the transaction with sequence number $predecessor(sn_t)$ and before the transaction with sequence number $successor(sn_t)$. We force transactions to commit according to the predefined order by inserting a conditional wait in *txn_commit*. When transaction t wants to commit, it waits until the transaction with sequence number $predecessor(sn_t)$ commits (line 8). To this end, transactions communicate via a sn_c object whose value is the sequence number of the last committed transaction (line 11).
Correctness. In the original OCC protocol correctness is guaranteed by atomically executing both the validation and write phase. However, the order in which active transactions execute those phases depends on their nondeterministic multithreaded execution. To conform with the predefined order the atomic block is replaced with a conditional

wait that restricts the order in which transactions are allowed to commit. Specifically, a transaction t that finishes its read phase is only allowed to perform the validation and write phases *after* the transaction that directly precedes t in the serial order has completed. Since transactions are totally ordered, *only one transaction at a time* can escape the conditional waiting on line 8. Correctness is maintained because the conditional wait also guarantees atomicity. The atomicity scope is between the wait condition (line 8) and updating sn_c (line 11).

3.2.2.3 Transaction modes

OCC employs a set of techniques to guarantee correctness, such as read and write sets, read set validation and deferred updates. With OCC *all* transactions are executed using the aforementioned techniques because *any* transaction *may* become the next transaction in the serialization order, which is being defined as transactions execute. Using such techniques imposes additional overhead when compared with an execution without any concurrency control.

However, unlike in OCC, in PCC the serialization order is predefined. Since PCC restricts the order in which transactions commit, they may now have to wait for their turn to commit, leading to a loss of parallelism. To mitigate this loss of parallelism, we make the key observation that at any moment there is always a single transaction, which we refer to as *fast*, which is the next transaction that is allowed to commit. We exploit the fact that the fast transaction is the next transaction allowed to commit to execute it without most concurrency control overheads. Hence, we distinguish between two types of transactions: fast and speculative. We describe both fast and speculative modes below.

Fast transaction. A fast transaction t is the *only* active transaction whose predecessors are all completed. A fast transaction is the next, and only, transaction allowed to commit. It can be executed more efficiently by merging the read and write phases and completely removing the validation phase, thus eschewing most of the traditional OCC techniques and associated overhead. Fast transactions execute according to the protocol in Figure 3.2c.

Read phase. Write operations no longer perform deferred updates; instead they use direct updates (line 7). Since updates are installed in place during the now combined read-write phase, read operations are reduced to simply reading the current object's value with no additional consistency checks or read set tracking (line 9).

Validation phase. Fast transactions are guaranteed to execute to completion without interference from other active transactions, thus the validation phase is unnecessary. (Transactions that switch on the fly to fast mode need to validate the speculative execution done up to that point; we elaborate below.)

Write phase. The write phase is implicitly executed during the read phase due to the direct update strategy, therefore the "write back" step is also completely eliminated.

Correctness. Our argument for correctness is the same as for the ordered commits technique. However a fast transaction does not speculatively perform the read phase and wait for its turn to transition to the validation and write phases. Instead the fast transaction executes the now combined read-write phase when it is already its turn to commit. A fast transaction is effectively given exclusive write permission to the shared state until it commits, so merging the read and write phases by replacing deferred with direct updates, and removing the validation phase, does not affect correctness.

Speculative transaction. A transaction whose turn to commit has not yet come is a speculative transaction, and it follows the ordered commit protocol (Section 3.2.2.2).

Live promotion. Since fast transactions bypass most concurrency control overhead, a live speculative transaction t, i.e. still executing its read phase, immediately switches to fast mode as soon as $sn_c = predecessor(sn_t)$ holds (line 1). Upon a live promotion, transaction t eagerly validates the portion of the read phase it has executed so far (line 2). If the validation is successful then t applies any pending writes to the shared state, *without* updating sn_c, and executes its remaining operations in fast mode (line 3). Otherwise t aborts and retries in fast mode (line 5).

Explicit aborts. If the transaction API has an explicit *txn_abort* operation to abort the current transaction, fast transactions must keep the write set as an undo log, i.e. remember the values they overwrite to restore them upon abort. The *txn_abort* operation may allow the developer to specify a “no retry” policy, i.e. abort the transaction without retrying it afterwards. If so, these “no retry” aborts must comply with the predefined order as they are equivalent to committing the current transaction as read only. This is done by processing a “no retry” explicit abort as a commit. For example, a speculative transaction waits for its turn, validates its read set, and updates sn_c if validation is successful, or retries if not. A fast transaction restores the write set (undo log) and updates sn_c.

Multiple simultaneous fast transactions. Multiple fast transactions can safely execute in parallel given additional knowledge about transactions. A string of successive transactions that do not have read-write nor write-write conflicts between themselves can all execute simultaneously as fast transactions, because the final outcome is independent of the order in which they commit. To implement multiple simultaneous fast transactions the runtime requires a compatibility matrix of all transactions. When a transaction becomes fast it publishes its information: transaction identifier, sequence number, and that it is active. Using this scheme, a transaction knows it can switch to fast mode if: (1) its predecessor is already fast (active or finished), and (2) it is compatible with all currently active fast transactions. If both conditions hold, the transaction can switch to fast mode.

3.3 Implementation

We implemented a Pot prototype consisting of an implementation of a sequencer and two concurrency control protocols: one where transactions execute using STM and another where transactions execute using HTM. Our sequencer implementation is generic and derives the transaction order from a round-robin thread order (Section 3.2.1). Next, we describe our STM (Section 3.3.1) and HTM (Section 3.3.2) implementations.

```
tm-begin($t$)
    $rv_t \leftarrow gv$
    acquire-fence
tm-write($t$, $addr$, $val$)
    add $(addr, val)$ to $W_t$
tm-read($t$, $addr$)
    if $addr \in W_t$
        return $W_t(addr)$
    $l_1 \leftarrow$ vlock($addr$)
    acquire-fence
    $value \leftarrow$ read($addr$)
    acquire-fence
    $l_2 \leftarrow$ vlock($addr$)
    if $l_1 > rv_t \vee l_1 \neq l_2$
        tm-abort
    add $addr$ to $R_t$
    return $value$
tm-commit($t$)
    for $(addr, -) \in W_t$ do
        if try-lock($addr$) fails
            tm-abort
    atomically
        $gv \leftarrow gv + 2$
        $wv_t \leftarrow gv$
    for $addr \in R_t$ do
        $l \leftarrow$ vlock($addr$)
        if $addr \in W_t$
            if version($l$) $> rv_t$
                tm-abort
        else if $l > rv_t$
            tm-abort
    for $(addr, val) \in W_t$ do
        write($addr, val$)
    release-fence
    for $(addr, -) \in W_t$ do
        set-and-unlock($addr, wv_t$)
```

a TL2.

```
tm-begin($t$)
    $rv_t \leftarrow gv$
    acquire-fence
    if first attempt
        $wv_t \leftarrow$ seq-no($tid$)
tm-write($t$, $addr$, $val$)
    add $(addr, val)$ to $W_t$
tm-read($t$, $addr$)
    if $addr \in W_t$
        return $W_t(addr)$
    $v_1 \leftarrow$ version($addr$)
    acquire-fence
    $value \leftarrow$ read($addr$)
    acquire-fence
    $v_2 \leftarrow$ version($addr$)
    if $v_1 > rv_t \vee v_1 \neq v_2$
        tm-abort
    add $addr$ to $R_t$
    return $value$
tm-commit($t$)
    wait until $gv = wv_t - 1$
    acquire-fence
    for $addr \in R_t$ do
        $v \leftarrow$ version($addr$)
        if $v > rv_t$
            tm-abort
    for $(addr, val) \in W_t$ do
        set-version($addr, wv_t$)
        release-fence
        write($addr, val$)
    release-fence
    $gv \leftarrow wv_t$
```

b Speculative PCC.

```
when $gv = wv_t - 1$
    acquire-fence
    for $addr \in R_t$ do
        $v \leftarrow$ version($addr$)
        if $v > rv_t$
            tm-abort
    for $(addr, val) \in W_t$ do
        set-version($addr, wv_t$)
        release-fence
        write($addr, val$)
tm-write($t$, $addr$, $val$)
    set-version($addr, wv_t$)
    release-fence
    write($addr, val$)
tm-read($t$, $addr$)
    return read($addr$)
tm-commit($t$)
    release-fence
    $gv \leftarrow wv_t$
```

c Fast PCC.

Figure 3.3: Transformation of TL2 into Pot Concurrency Control (PCC). Figure 3.3a shows the original TL2 transaction. Figures 3.3b and 3.3c show a PCC transaction in speculative and fast mode, respectively.

3.3.1 Software Transactional Memory

The ordered commits technique ensures that only one transaction executes its commit procedure at a time. In NOrec [16] commits are also sequential. While this similarity makes NOrec a potential baseline for Pot, NOrec eschews per-memory location metadata and uses value-based validation instead. Consequently, speculative transactions are unable to identify which particular memory location is written when the fast transaction performs a write. As such, implementing fast transactions while still preserving opacity would require that, every time a fast transaction performs a write, all speculative transactions would have to validate their entire read set, regardless of which specific memory location was written by the fast transaction. Instead, our Pot STM protocol is based on TL2 [20], a popular STM that uses per-memory location metadata, so that speculative transactions do not have to perform incremental validation on reads.

Baseline STM transaction. In a nutshell, TL2 works as follows. There is a global version and a table of versioned locks, i.e., a version and a lock bit implemented as a single value—vlocks for short. Odd versions are locked and even versions are unlocked. Each memory address is mapped to one vlock. When a transaction starts, it samples the global version gv to rv_t and performs an acquire fence (Figure 3.3a, lines 2–3). The transaction can safely read any value whose version is less than or equal to its rv_t sampling. The fence with acquire semantics ensures that this transaction observes all the memory writes performed by the transaction that updated gv's value to rv_t. Write operations are buffered in the write set (line 5). Read operations return the value of a buffered write if there is any (line 7–8). Otherwise, they perform a consistent read by: (1) reading the address' vlock to $v1$ (line 9), (2) performing an acquire fence (line 10), (3) reading the memory address (line 11), (4) performing another acquire fence (line 12), and (5) reading the vlock again to $v2$ (line 13). The first fence ensures that the memory address value is at least as recent as $v1$. (If $v1$ is 42, then the value read has version 42 or newer.) The second fence ensures that if the value is newer than $v1$, then $v2$ is at least as recent as the value's version. (If the value read has version 43, $v2$ is 43 or newer.) If $v1$ is not locked, and $v1 \leq rv_t$, and $v1 = v2$, then the read successfully returns a consistent value; otherwise, the transaction aborts (lines 14–17).

The commit operation locks every address in the write set by performing a compare-and-swap on their vlocks. If any of the compare-and-swap operations fails, then the transaction releases any acquired locks and aborts (lines 19–21). After successfully acquiring the vlocks, the transaction performs an atomic add-and-fetch by 2 on gv and stores gv's new value in wv_t (line 22). Then, the transaction validates its read set by checking whether all memory addresses read are unlocked and their version is still compatible with rv_t. If any check fails then the transaction restores any acquired locks and aborts (lines 25–31). Note that the atomic add-and-fetch operation ensures that: (1) any other transaction that starts meanwhile and observes $gv = wv_t$ will at least observe all the write set vlocks as acquired, and (2) if any transactions committed since this transaction started, i.e. $wv_t > rv_t + 2$, and wrote to a memory address read by this transaction, then the read set validation will observe vlocks as locked or with a version newer than rv_t.

At this point the transaction successfully commits. It writes back any buffered writes, performs a release fence, and unlocks the write set, setting every vlock to wv_t. The release fence ensures that if any transaction observes a vlock with version wv_t then it also observes the value written by the transaction.

Speculative STM transaction. To implement PCC, we leverage the fact that TL2 uses a global version and retrofit sequence numbers directly as versions. Thus, transactions communicate the commit order via gv. A consequence of ordered commits is that we no longer require locks, just versions, as they were only needed due to concurrent commits.

When a transaction starts for the first time, it requests its sequence number wv_t from the sequencer by supplying the thread's identifier tid (Figure 3.3b, lines 4–5). Read operations are similar to TL2 except that we no longer test if the address is locked (line 16). When the transaction attempts to commit, if necessary it waits until $gv = wv_t - 1$ (line 21). Once $gv = wv_t - 1$, we perform an acquire fence that ensures that the following read set validation observes the newest version of the addresses read (line 23–26). In the write back step, we first update the address' version, perform a release fence, and then write the new value (lines 27–30). As discussed before, the release fence ensures that if any transaction observes the written value, it also observes the new version number. Finally, the transaction updates gv, signaling the next transaction that it is its turn to commit (line 32). The update of gv is preceded by a release fence to ensure that all transactions that see the new value of gv will also see the new values for the objects written in the write back.

Fast STM transaction. The fast mode write operation is equivalent to the write back step of a speculative transaction, i.e. updates the version number, performs a release fence, and writes the new value (Figure 3.3c, lines 12–14). The read operation is reduced to a regular load from memory (line 16), and the commit operation simply updates gv (line 19).

Live promotion. A speculative STM transaction t changes to fast on the fly when it detects that it is its turn, i.e. $gv = wv_t - 1$ (Figure 3.3c, lines 1–10). In our implementation we check whether the condition holds whenever the speculative transaction begins, reads, or writes.

3.3.2 Hardware Transactional Memory

Implementing PCC in HTM poses unique challenges when compared with an STM implementation. Existing HTMs use the cache to maintain the read and write set, and rely on the cache coherence protocol to detect conflicts. HTMs are also *best effort*, i.e., hardware transactions are not guaranteed to eventually commit, even in the absence of conflicts, be it because the transaction's footprint exceeds the cache capacity, or due to the execution of an illegal instruction, an interrupt, a page fault, etc. Therefore, we must always provide a software fallback to guarantee progress. These characteristics pose three challenges, namely: (a) how to ensure that transactions eventually progress, (b) how to implement ordered commits without inducing false conflicts, and (c) how to implement fast transactions.

In our prototype we ensure progress using the most common fallback that achieves opacity: resorting to a global lock. Every time a transaction acquires the global lock, all hardware transactions abort and only retry when the lock is released.

In HTM, the commit operation is implemented entirely in hardware. This poses a challenge on how to implement ordered commits because we introduce conflicts if transactions signal each other whose turn it is to commit using a shared variable. For example,

```
tm-begin($t$)
  if $first\ attempt$
    $path_t \leftarrow$ HW
    $tries_t \leftarrow 10$
  wait while locked($gl$)
  tbegin
  if locked($gl$)
    tm-abort
  execute app. code
tm-abort($t$)
  if $persistent$
    $tries_t \leftarrow 0$
  else
    $tries_t \leftarrow tries_t - 1$
  if $tries_t = 0$
    lock($gl$)
    $path_t \leftarrow$ SW
    execute app. code
tm-commit($t$)
  if $path_t = HW$
    tcommit
  else
    unlock($gl$)
```

a HTM.

```
tm-begin($t$)
  if $first\ attempt$
    $path_t \leftarrow$ HW
    $tries_t \leftarrow 10$
    $sn_t \leftarrow$ seq-no($tid$)
  wait while locked($gl$)
  if $sn_c = sn_t - 1$
    tbegin($ROT$)
    switch to fast mode
  else
    $sn \leftarrow sn_c$
    tbegin
    if locked($gl$)
      tm-abort
  execute app. code
tm-abort($t$)
  if $persistent$
    wait until $sn_c = sn_t - 1$
  else
    wait until $sn_c > sn$
tm-commit($t$)
  tsuspend
  wait until $sn_c = sn_t - 1$
  tresume
  tcommit
  $sn_c \leftarrow sn_t$
```

b Speculative PCC.

```
tm-begin($t$)
  tbegin($ROT$)
  execute app. code
tm-abort($t$)
  if $persistent$
    $tries_t \leftarrow 0$
  else
    $tries_t \leftarrow tries_t - 1$
  if $tries_t = 0$
    lock($gl$)
    $path_t \leftarrow$ SW
    execute app. code
tm-commit($t$)
  if $path_t =$ HW
    tcommit
  else
    unlock($gl$)
  $sn_c \leftarrow sn_t$
```

c Fast PCC.

Figure 3.4: Transformation of Hardware Transactional Memory (HTM) into Pot Concurrency Control (PCC). Figure 3.4a shows the original HTM transaction. Figures 3.4b and 3.4c show a PCC transaction in speculative and fast mode, respectively.

imagine two non-conflicting transactions t_1 and t_2, serialized in that order. Transaction t_2 attempts to commit before t_1. It reads the commit-order variable, sn_c, and observes that it is still not its turn so it waits, e.g., because t_2 can only commit when $sn_c = 1$. When transaction t_1 commits it sets $sn_c = 1$, triggering a conflict in t_2 because it observed a (now) stale value. Implementing fast transactions is also challenging because all concurrency control is performed by the hardware.

To implement our prototype we looked at the existing HTMs from Intel [72] and IBM [14]. To implement ordered commits without inducing false aborts we require the possibility to perform non-transactional accesses, i.e. that do not trigger transactional conflicts. Unfortunately, Intel provides no support for non-transactional accesses. However, IBM's HTM has two instructions, `tsuspend` and `tresume`, that allow the possibility to suspend, and resume, transactional execution inside a hardware transaction. (While in suspended mode accesses are performed non-transactionally.)

IBM's HTM also provides a special kind of transaction called Rollback-only Transaction (ROT). According to IBM, ROTs are intended to be used for single thread algorithmic speculation [14]. For this reason, ROTs also buffer transactional writes in the cache but do not maintain a read set. Furthermore, ROTs do not observe buffered transactional writes from other transactions and all writes performed by a ROT become visible to other transactions atomically, making them a prime choice to implement fast transactions. However, note that ROTs may nevertheless abort due to write-write conflicts with other concurrent transactions. For these reasons we implemented our prototype on IBM's HTM. It is still possible to implement Pot with Intel's HTM, albeit with ordered commits inducing false aborts and fast transactions being regular transactions.

Baseline HTM transaction. The relevant IBM's HTM instructions are `tbegin` and `tcommit`, to start and commit a hardware transaction, respectively. We initialize two important variables, $path_t$ and $tries_t$, when the application starts a transaction, by invoking our usual *txn_start* operation. $path_t$ is either HW or SW depending on whether the transaction will execute as a hardware transaction or by software using the global lock fallback. $tries_t$ holds the number of remaining attempts to execute the transaction in hardware until we fallback to software (Figure 3.4a, lines 2–4). (We retry 10 times like in GCC's experimental implementation.) If there is an ongoing transaction executing in software, we wait until the global lock is free, otherwise the hardware transaction may observe an inconsistent state and violate opacity (line 5). After the lock is free we start a hardware transaction by issuing the `tbegin` instruction (line 6). From this point on, every memory access is performed transactionally. Finally, we subscribe the global lock by checking if it is locked before proceeding with the actual application code (lines 7–8). By checking if the lock is taken it becomes part of the transaction's read set, so if any transaction falls back to software, any active hardware transaction is immediately aborted.

Committing a transaction depends on whether it executed in hardware ($path_t$ = HW) or software (SW). We commit a hardware transaction using the `tcommit` instruction, whereas for a software transaction we simply release the global lock (lines 20–23). Note that the `tcommit` operation may still trigger an abort if the transaction fails to commit.

When a hardware transaction aborts, the control flow jumps to the *txn_abort* handler. First, we check whether the abort is expected to be persistent by inspecting the IBM's TEXASR register, which contains several hints about the reason why the transaction aborted. For example, an abort due to capacity restrictions is persistent. If the abort is persistent, we fallback to software by acquiring the global lock and execute the transaction's application code (lines 11, 15–18). Otherwise, we decrement the number of remaining attempts and control flow jumps back to *txn_start*.

Speculative HTM transaction. Like in our STM implementation, when a transaction starts for the first time it requests a sequence number from the sequencer (Figure 3.4b, line 5). After waiting until the global lock is free, we check whether it is the transaction's turn to commit. If so, we begin a ROT and switch to fast mode (lines 8–9). Otherwise, we sample the sequencer number of the current fast transaction *sn* (we explain why shortly), then begin a hardware transaction, and subscribe to the global lock (lines 11–13). To commit a transaction we: (1) issue the `tsuspend` instruction to suspend transactional execution (line 22), (2) wait for our turn to commit (line 23), (3) issue the `tresume` instruction to resume transactional execution (line 24), and (4) issue the `tcommit` instruction to commit (line 25). If the commit is successful we update the sn_c variable accordingly (line 26).

If the speculative transaction aborts due to persistent reasons, there is no point in retrying the transaction until it is it's turn to commit (line 18). Otherwise, we wait until the concurrent fast transaction commits to retry the speculative transaction (line 20)—recall that we sampled its sequence number sn when we started the aborted hardware transaction (line 11). The rationale for waiting for the concurrent fast transaction to commit is to minimize the chances of aborting the fast transaction via write-write conflicts.

Fast HTM transaction. As previously stated, fast transactions execute as ROTs. Unlike regular hardware transactions, ROTs do not maintain a read set so they enjoy an increased capacity limit that can be used exclusively for writes. Transactions that previously exceeded capacity constraints and had to fallback to software might now be able to commit in hardware. This has the potential to increase the parallelism in the system because falling back to software effectively "stops the world."

Committing a fast transaction is essentially equivalent to the standard HTM transaction with an additional update to sn_c (Figure 3.4c lines 14–18). If a fast transaction aborts due to capacity restrictions it falls back to software (lines 5–6, 9–12).

Note that the hardware ensures that the fast transaction's reads do *not* observe the writes of concurrent speculative transactions. Moreover, if the fast transaction reads a memory location that has been written by a concurrent speculative transaction, the hardware aborts the speculative transaction immediately if it is executing, or when it issues the `tresume` instruction if it is suspended.

3.4 Experimental evaluation

All experiments were run on a 10-core IBM POWER8 with a total of 128GB RAM. We highlight the fact that the machine has a NUMA architecture. Particularly, the memory latencies in our experiments are as follows: with 1 to 4 threads memory latencies are uniform, while with 8 or more threads memory latencies increase up to 2×.

We evaluate Pot using the popular STAMP 0.9.10 benchmark suite [50] and STMBench7 [31], using the parameters listed in Figure 3.5. STAMP consists of 8 representative applications from different domains, e.g. online transaction processing, iterative clustering algorithms, and Delaunay mesh refinement (Vacation, KMeans, and Yada, resp.) [50]. Some STAMP benchmarks, such as Labyrinth, KMeans, and Yada, output non-deterministic results using STM. The benefits of Pot in these benchmarks are that the computed Labyrinth's solution, KMeans' clusters, and Yada's mesh, are always the same across executions. STMBench7 is a more complex benchmark suggestive of CAD, CAM or CASE software [31]. Results are the average of five runs. The GCC version is Red Hat 5.1.1-4.

Benchmark	Parameters
Bayes	`-v 32 -r 4096 -n 10 -p 40 -i 2 -e 8 -s 1`
Genome	`-g 65536 -s 32 -n 16777216`
Intruder	`-a 10 -l 2048 -n 8192 -s 1`
Kmeans–	`-m 40 -n 40 -t 0.00001 -i inputs/random-n65536-d32-c16.txt`
KMeans+	`-m 15 -n 15 -t 0.00001 -i inputs/random-n65536-d32-c16.txt`
Labyrinth	`-i inputs/random-x512-y512-z7-n512.txt`
SSCA2	`-s 20 -i 1.0 -u 1.0 -l 3 -p 3`
Vacation–	`-n 8 -q 90 -u 98 -r 1048576 -t 4194304`
Vacation+	`-n 8 -q 10 -u 90 -r 1048576 -t 4194304`
Yada	`-a 15 -i inputs/ttimeu1000000.2`
STMBench7 (r)	`-t true -w r`
STMBench7 (rw)	`-t true -w rw`
STMBench7 (w)	`-t true -w w`

Figure 3.5: Parameters used in STAMP and STMBench7.

3.4.1 Software Transactional Memory

In this section we evaluate our Pot STM prototype. We seek to answer the following questions:

Are fast transactions effective? (Section 3.4.1.1.) Yes, they successfully reduce concurrency control overheads and execute faster than regular transactions. Our experiments show that fast transactions already execute faster than regular transactions even when they perform as little as 1 read and 1 write access, despite the addition work performed regarding the sequencer and switching modes (Figure 3.6).

Does Pot ensure determinism efficiently? (Section 3.4.1.2.) We argue that it does. Our experiments show that Pot ensures deterministic execution across all of STAMP's benchmarks with an average slowdown over nondeterministic execution of less than 2× (geometric mean of Figure 3.7), and it is ≈ 5× *faster* on average than the nondeterministic baseline in STMBench7 (geometric mean of Figure 3.10).

Does Pot improve upon the state of the art? (Section 3.4.1.2.) Yes, Pot successfully lowers the overheads of ensuring determinism when compared with DeSTM [56]. Our experiments show that, when compared to DeSTM, Pot is up to ≈ 3× faster than DeSTM on average across the STAMP benchmarks (geometric mean of Figure 3.7) and up to ≈ 9× faster on average in STMBench7 (geometric mean of Figure 3.10), and scales better with the number of threads (Figure 3.11 and 3.12).

3.4.1.1 Effectiveness of fast transactions

The fast transaction's objective is to reduce concurrency control overheads in order to mitigate the potential loss of parallelism introduced by ordered commits. To measure how effective is the fast execution mode we executed a microbenchmark that consists of a simple key-value data structure implemented with an array of counters. We use a single thread, and vary the number of accesses performed by transactions, and the accesses' read/write ratio.

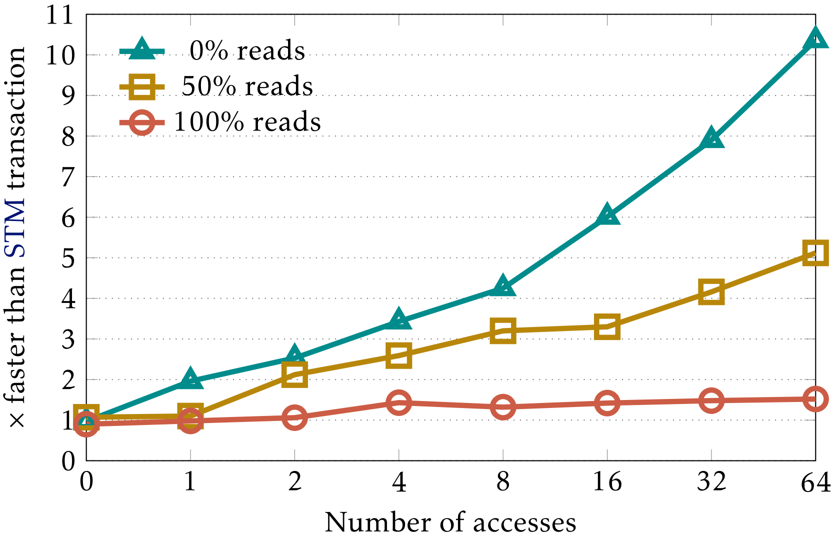


Figure 3.6: Speedup achieved by a Pot fast transaction over the baseline STM transaction.

Figure 3.6 shows how much faster the Pot fast transaction protocol is than the baseline STM transaction. Transactions with zero accesses consist of *txn_begin* immediately followed by *txn_commit*. This allows us to measure the overhead imposed by the additional work performed by the sequencer, ordered commits, and transaction modes, which is negligible. By increasing the number of accesses we observe that, as expected, fast transactions perform increasingly better than the baseline. We also observe that write operations contribute more to the achieved speedup. This is due to the fact that in the baseline STM write operations impose overhead on reads because reads must query the write set for possible buffered values. Write operations also impose overhead on the commit operation due to the need to lock the write set, perform the write back, and unlock the write set. Fast transactions bypass all these sources of overhead. However, fast transactions do not achieve observable gains when transactions are read-only. This is because read-only transactions in the baseline STM do not need to validate the read set at commit time—they are serialized at begin time. Overall, fast transactions are successful in minimizing concurrency control overheads, even for transactions that perform as little as one read and one write.

3.4.1.2 Comparison with the state of the art

In this section we evaluate deterministic execution using Pot in the popular STAMP 0.9.10 benchmark suite [50], and STMBench7 [31]. We also compare Pot against DeSTM, a state of the art system in deterministic execution of STM programs. (Please refer to section 6.1 for a more in-depth description of DeSTM and its comparison to Pot.) To perform an apples-to-apples comparison, we implemented DeSTM in our own prototype.[3] Both Pot

[3] DeSTM is not publicly available. We asked the authors for the source code via e-mail but got no response.

and DeSTM are based on the same baseline STM protocol and use the exact same sequencer. We also implemented a deterministic and non-speculative solution based on a global lock that transactions acquire according to the order defined by the sequencer, i.e. transactions acquire a global lock at *txn_begin* and release it at *txn_commit* (PoGL, as in Preordered Global Lock). The rationale is that PoGL is a "trivial" implementation of PCC without any speculation. We show results for DeSTM, PoGL, ordered commits only (Pot−), ordered commits and transaction modes (Pot∗), and ordered commits, transaction modes, and live promotion (Pot).

Performance. Figure 3.7 quantifies the cost of deterministic multithreading when using DeSTM, PoGL, and Pot, on STAMP. With it we seek to answer the following question: "How much slower is the execution with x threads if we want determinism?" The Figure reports the execution time normalized to the baseline nondeterministic STM execution (y axis) of every benchmark of the STAMP suite, when executed with DeSTM, PoGL, Pot− (ordered commits), Pot∗ (ordered commits and transaction modes) and Pot (ordered commits, transaction modes, and live promotion) using from 2 to 16 threads (x axis). In these plots lower is better, and values below 1 mean that the deterministic execution was *faster* than standard nondeterministic execution. Four observations stand out: (a) the cost of ensuring determinism increases with the number of threads, (b) Pot outperforms DeSTM in all benchmarks, (c) Pot is at most ≈ 3× slower than the nondeterministic baseline, while DeSTM suffers from a slowdown of up to ≈ 11×, (d) Pot is even *always faster* than the baseline STM execution on Genome, and (e) although PoGL works well in some workloads, Pot achieves the best of both worlds: Pot is comparable to PoGL on the workloads PoGL works well, and considerably outperforms PoGL on the remaining workloads (e.g. ≈ 2.5× on Intruder, ≈ 3× on Labyrinth and Vacation+, and ≈ 5× on Vacation−).

The fact that the cost of ensuring determinism increases with the number of threads is unsurprising; the probability of a transaction t attempting to commit before its turn increases with the number of threads, particularly if there are transactions ordered before t that take longer than t. Pot's ordered commits and transaction modes minimize these situations to increase the probability of transactions not having to wait for their turn to commit. Figure 3.8 supports this claim. It shows, for each benchmark/thread combination, how much time DeSTM transactions "waste" to enforce determinism, on average, when compared to Pot. We can observe that in general DeSTM transactions spend more time waiting for their turn to commit. Figure 3.9 shows two example scenarios that highlight the differences between DeSTM and Pot. In DeSTM time is divided into rounds, and in each round each thread executes one transaction. A transaction cannot start if some transaction from the previous round has not finished yet (Figure 3.9a), and cannot commit, even on its turn, if some transaction from the same round has not started yet (Figure 3.9b). In contrast, Pot realizes that rounds are not necessary to respect a predefined serial order, so transactions never wait to start, nor to commit on their turn.

Pot also accelerates the execution of the next transaction to commit according to the

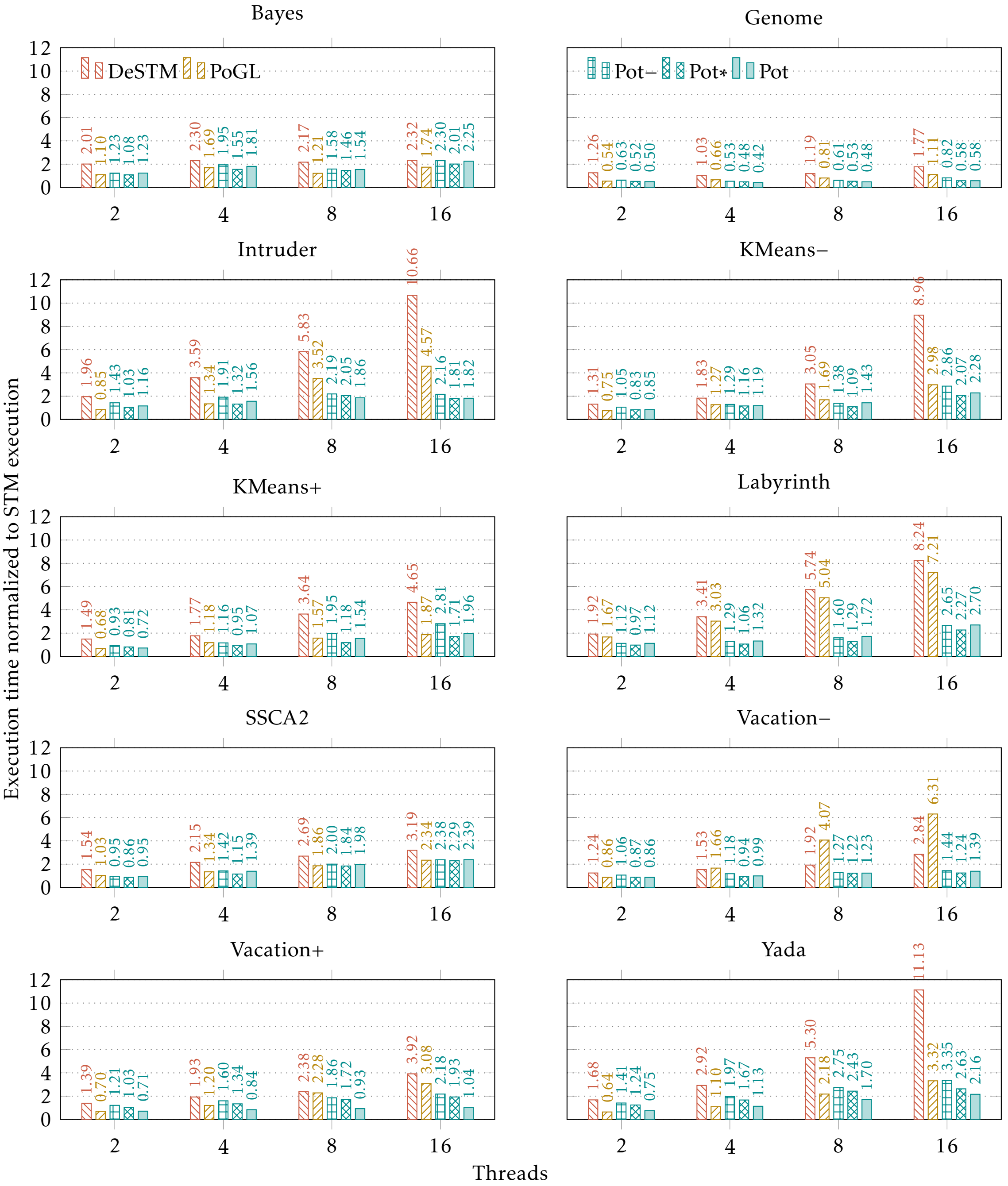
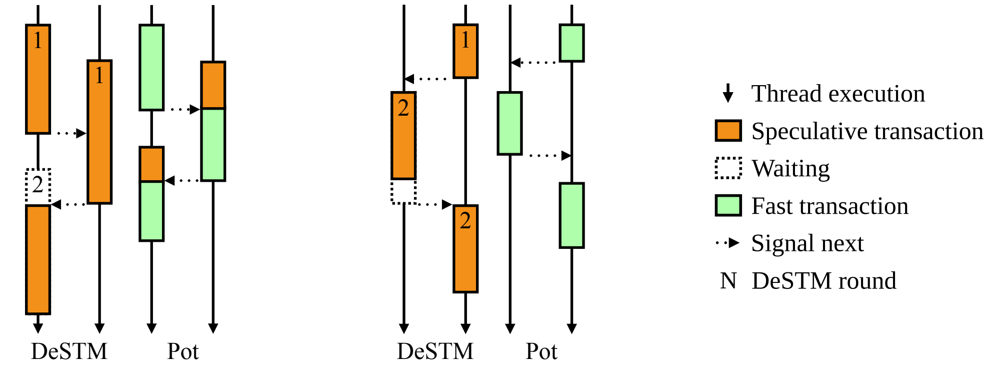


Figure 3.7: **How much slower is the execution of each STAMP benchmark with *x* threads if we want determinism?** The y axis measures the execution time using DeSTM, PoGL (preordered global lock), Pot− (ordered commits), Pot∗ (ordered commits and transaction modes), and Pot (ordered commits, transaction modes, and live promotion), normalized to the nondeterministic execution using the baseline STM (lower is better). − and + refer to the relative levels of contention in the configuration.

serial order. From Figure 3.6 we deduce that the benefits of the fast mode should be more apparent in benchmarks with bigger transactions with higher write-to-read ratio, and/or higher contention. Fast transactions (Pot∗) further improve performance over ordered

Benchmark	Threads			
	2	4	8	16
Bayes	1.88×	1.03×	0.95×	0.68×
Genome	3.24×	3.99×	3.92×	3.24×
Intruder	2.92×	3.11×	2.19×	1.74×
Kmeans−	4.16×	2.54×	2.22×	1.68×
KMeans+	3.62×	2.76×	1.97×	1.16×
Labyrinth	6.61×	5.31×	2.67×	0.77×
SSCA2	4.29×	1.62×	1.36×	1.34×
Vacation−	5.52×	4.01×	3.51×	3.60×
Vacation+	5.91×	4.93×	4.41×	5.29×
Yada	3.00×	3.26×	2.23×	1.69×
STMBench7 (r)	2.90×	3.53×	2.34×	4.73×
STMBench7 (rw)	8.02×	5.96×	7.16×	7.82×
STMBench7 (w)	15.10×	11.77×	9.17×	11.31×

Figure 3.8: Time that DeSTM transactions spend waiting to enforce determinism compared to Pot, using the STAMP and STMBench7 benchmarks. A value of 2× means that, on average, DeSTM transactions spend 2× more time waiting for their turn (hence higher is better for Pot).



a DeSTM waits to start.

b DeSTM waits to commit.

Figure 3.9: Examples of the difference between DeSTM and Pot. In DeSTM time is divided into rounds, and in each round each thread executes one transaction. A transaction cannot start if some transaction from the previous round has not finished yet (a), and cannot commit, even on its turn, if some transaction from the same round has not started yet (b). In contrast, Pot realizes that rounds are not necessary to respect a predefined serial order, so transactions never wait to start, nor to commit on their turn. Pot also accelerates the execution of the next transaction to commit according to the serial order.

commits in all benchmarks (Figure 3.7). However, the overhead of our implementation of live promotion (Pot) only pays off in Genome, Vacation+, and Yada.

We also experimented with STMBench7. Figure 3.10 shows the throughput of DeSTM, PoGL, Pot−, Pot∗, and Pot, normalized to the throughput of the baseline STM. Because STMBench7 features a more diverse set of transaction profiles, with more complex read-write transactions, live promotion is very effective at boosting Pot's throughput: in fact,

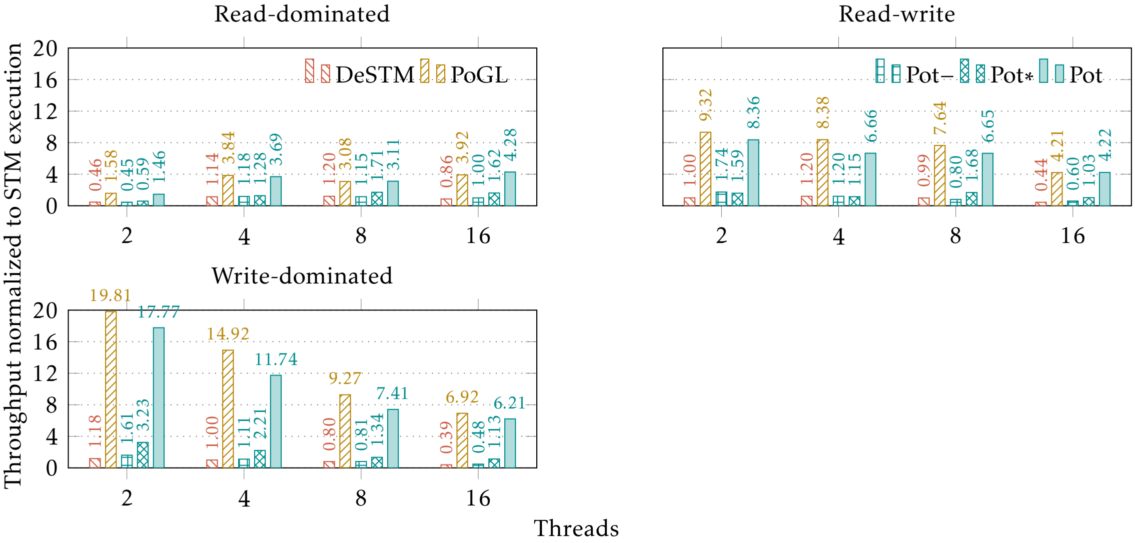


Figure 3.10: **How much faster is the execution of STMBench7 with *x* threads if we want determinism?** The y axis measures the throughput using DeSTM, PoGL (preordered global lock), Pot− (ordered commits), Pot∗ (ordered commits and transaction modes), and Pot (ordered commits, transaction modes, and live promotion), normalized to the nondeterministic execution using the baseline STM (higher is better). The titles indicate the workload type.

Pot is *always faster* than the nondeterministic baseline, usually by more than 3×.

To conclude, in our experiments Pot is a good general solution because it achieves the best of both worlds: when speculation is effective Pot provides superior performance, and when speculation is not effective Pot's performance is very close to PoGL's (Figures 3.7 and 3.10). Pot's excellent results compared to DeSTM's are explained by both the speedups that fast transaction can achieve, as observed in Figure 3.6, and the decrease of the time transactions spend waiting for their turn, as we observe in Figure 3.8. Pot marks a significant advance over the state of the art in performance, and provides promising evidence that using both STM and determinism to enable multithreaded replicas for fault tolerance, and/or to ease multithreaded programming, may be practical.

Scalability. We further evaluate Pot's scalability compared to a singlethread execution using the baseline STM on STMBench7 and all of the STAMP benchmarks. For comparison we also show results for DeSTM and the baseline STM itself. The baseline's behavior serves as a guide for what to expect from Pot and DeSTM's implementation: we don't expect them to scale if the baseline does not scale. However, ideally we should expect the Pot and DeSTM implementation to scale, even if shyly, despite the overheads required to ensure determinism, particularly the need to wait to enforce the deterministic commit order. Figures 3.11 and 3.12 show the results for STAMP and STMBench7, respectively. We observe that DeSTM fails to scale, whereas Pot is able to scale up to some point, notably in Genome, Intruder and Vacation. Pot shows better results than the baseline on STMBench7 because Pot inherently provides stronger progress guarantees to the more

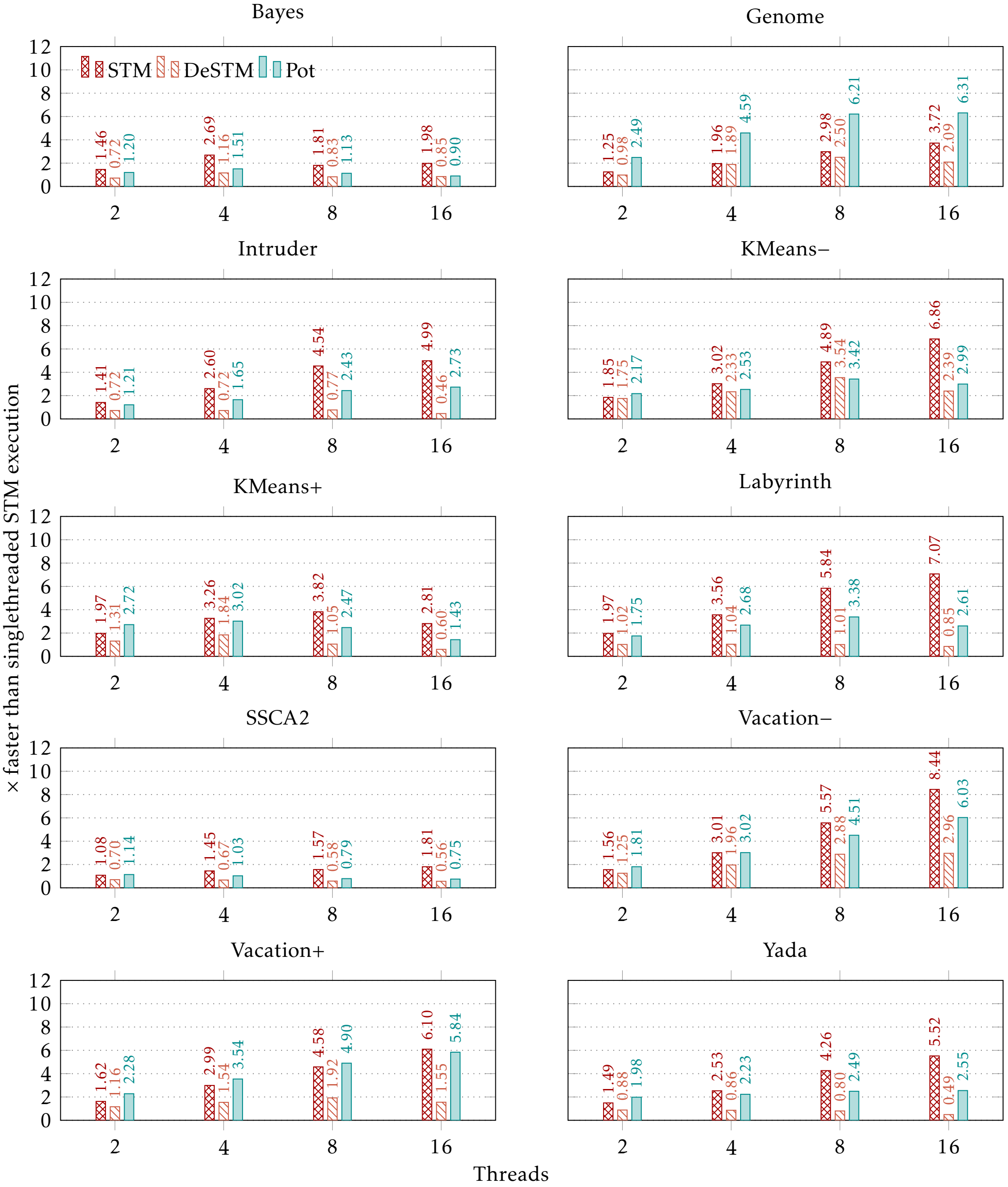


Figure 3.11: Scalability of deterministic execution using DeSTM and Pot on STAMP. The y axis measures the speedup over a singlethread baseline STM execution. A value of 1 means the execution time was the same as the baseline, a value greater than 1 means the execution time was faster (better), and a value less than 1 means the execution time was slower (worse).

complex transactions in the benchmark: while they struggle to commit in the baseline STM, in Pot they eventually do when it is their turn, and even have their execution sped up by the fast mode.

As threads increase it becomes increasingly challenging to mask the overhead required

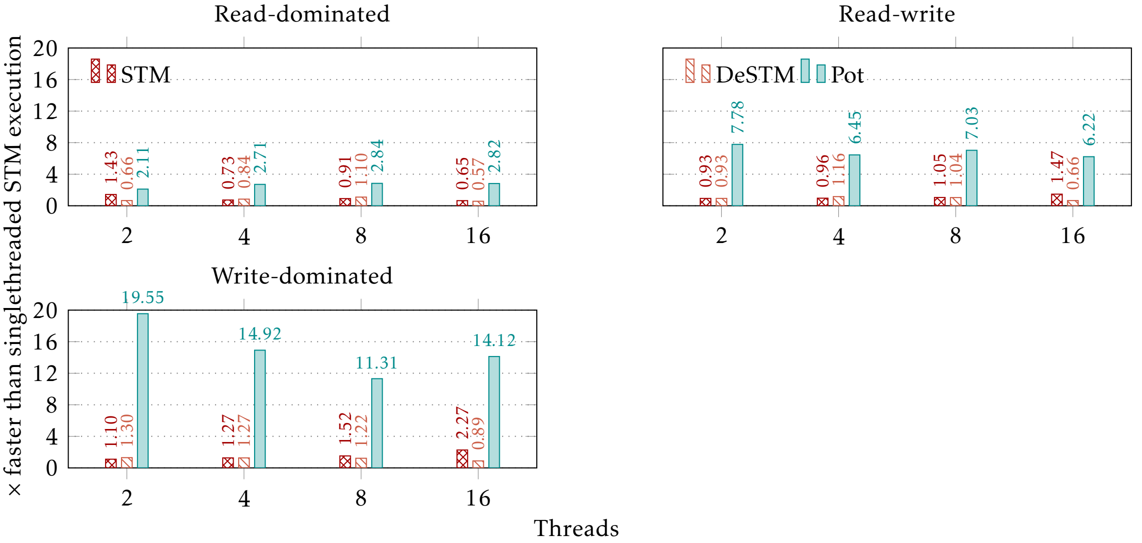


Figure 3.12: Scalability of deterministic execution using DeSTM and Pot on STMBench7. The y axis measures the speedup over a singlethreaded baseline STM execution. A value of 1 means the throughput was the same as the baseline, a value greater than 1 means the throughput was greater (better), and a value less than 1 means the throughput was lower (worse). The titles indicate the workload type.

to ensure determinism, but nonetheless Pot manages to keep up with the baseline up to a point. As part of future work we plan to address this issue by taking advantage of commutativity: if two successive transactions in the predefined serial order commute they can both execute simultaneously as fast transactions. The knowledge of whether two transactions commute can either be fed by the programmer via some sort of annotations, or inferred via analysis.

3.4.2 Hardware Transactional Memory

We also evaluate our Pot HTM implementation using the STAMP benchmark suite. We are interested in answering the following questions: (1) how effective are fast transactions, and (2) what is the cost that Pot incurs in to ensure deterministic execution.

Are fast transactions effective? (Section 3.4.2.1.) Yes, Pot fast transactions enjoy increased capacity limits when compared to regular transactions. Our experiments show that for 4 of the STAMP benchmarks, Pot fast transactions greatly reduce the need to fall back to software (Figure 3.13).

What is the cost that Pot incurs in to ensure determinism? (Section 3.4.2.2.) Our experiments show that Pot ensures deterministic execution across all of STAMP's benchmarks with moderate overhead (Figure 3.14.)

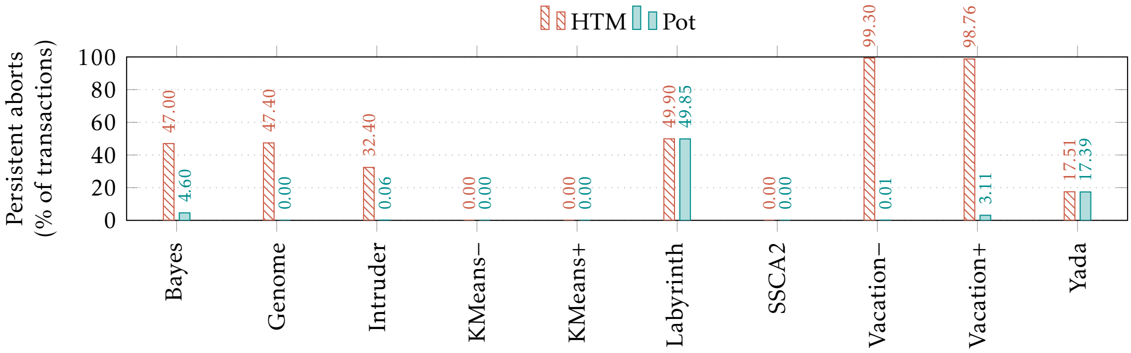


Figure 3.13: Percentage of transactions that experience persistent aborts using baseline HTM transactions and Pot fast HTM transactions on the STAMP benchmarks (lower is better). − and + refer to the relative levels of contention in the configuration.

3.4.2.1 Effectiveness of fast transactions

While our Pot STM fast transaction is able to reduce concurrency control overheads, implementing a HTM fast transaction that effectively reduces concurrency control overheads would require hardware support that is currently unavailable in existing processors. However, by exploiting IBM's Rollback-only Transactions (ROTs), Pot HTM fast transactions enjoy increased capacity limits, which increases the chance of committing more transactions entirely in hardware without falling back to the global lock.

We executed each benchmark with regular HTM and Pot using a single thread. Since there is only one thread executing there are no aborts due to concurrency, however transactions may still abort spuriously; thus we only count aborts that the hardware hints to be persistent—we collect this information from the TEXASR register as we discuss in Section 3.3.2. Figure 3.13 shows that the transactions that the baseline HTM cannot accommodate in both Labyrinth and Yada are also not accommodated by Pot's fast transaction. The transactions of KMeans and SSCA2, on the other hand, can all execute without problem. The rest of the benchmarks have a mix of transactions that can and cannot execute in hardware. In these we can clearly see the benefit of Pot's fast transactions: for example, in Bayes around 47% of the transactions can not be accomodated by the baseline HTM but this number falls to around 5% with Pot. Indeed, with Pot the number of transactions that are not accomodated by the hardware falls down from more than 30% to less than 5%. This means that Pot fast HTM transactions are successful at avoiding to fall back to the global lock. Thus, fast transactions manage to regain some of the parallelism lost to ordered commits when the baseline HTM falls back to software.

3.4.2.2 Performance

Figure 3.14 shows the overhead of deterministic multithreading using Pot HTM. It has less overhead on benchmarks where the baseline often falls back to software (Bayes, Genome, Vacation). In Genome Pot always outperforms the nondeterministic execution.

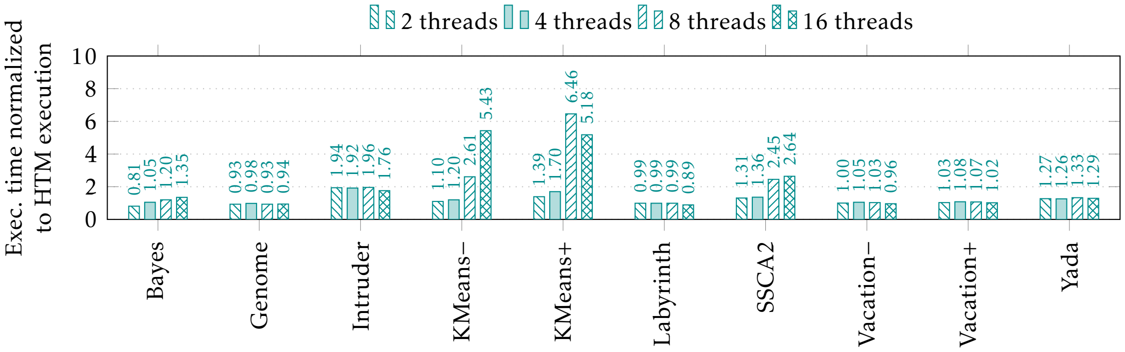


Figure 3.14: Deterministic execution of STAMP using Pot. The y axis measures the execution time normalized to the nondeterministic execution using the baseline HTM (lower is better). − and + refer to the relative levels of contention in the configuration.

Vacation's results in Figure 3.14 may seem unintuitive given than the baseline HTM practically always falls back to the global lock while Pot mostly executes without resorting the global lock (Figure 3.13). However, note that since all transactions executing in speculative mode exceed the hardware capacity, Pot is also executing one transaction at a time, albeit in fast mode instead of needing to fall back to the global lock.

Arguably the more interesting benchmarks are the ones where the baseline performs well, i.e. falls back less to the global lock (Intruder, KMeans, SSCA2, and Yada from Figure 3.13). In Intruder and Yada, in Figure 3.14, Pot achieves modest overheads of up to 2×. KMeans and SSCA2 are optimal for the baseline HTM, featuring small transactions with few accesses and conflicts. These characteristics make it difficult to mask the overheads of ensuring determinism. KMeans also features an abundant use of thread synchronization via barriers which amplifies the overhead caused by the sequencer while assigning sequence numbers deterministically. Also note that since fast transactions are not sped up in HTM, there is a noticeable drop in performance from 4 to 8 threads and even more from 8 to 16 threads due to the increased memory latencies of the NUMA architecture and hardware oversubscribing.

To the best of our knowledge, Pot advances the state of the art by enabling deterministic execution of HTM-based multithreaded programs for the first time. Overall, Pot achieves deterministic execution with lower overhead at lower thread counts, but increased memory latencies lead to a drop in performance relatively to the nondeterministic baseline. Efficiently achieving deterministic execution in the presence of non-uniform memory accesses represents an interesting future research avenue. The results achieved by Pot STM fast transactions suggest that hardware support for fast transactions that do not abort due to conflicts with other transactions may be worthwhile.

Chapter 4

LSD: Lazy State Determination

In this chapter, we present Lazy State Determination (LSD), a new interface to express transactions that allows the database to collect semantic information useful to achieve higher performance under contention without sacrificing safety. We also describe new optimistic and pessimistic concurrency control algorithms for providing (strict) serializability while exploring semantic information to increase concurrency in the presence of contention, using novel *condition validation* and *condition locking* techniques. Finally, we evaluate our implementation of an LSD prototype using the TPC-C benchmark [64] and microbenchmarks.

The chapter is organized as follows. We start by introducing and motivating the need for LSD in Section 4.1. We proceed with an overview of LSD in Section 4.2, by explaining the issues of the standard interface through a motivating example, from which we derive the LSD interface. Section 4.3 then presents LSD in detail, describing the techniques we use to design LSD-aware variants of OCC and 2PL in Section 4.3.3, and how to adapt 2PC for distributed LSD transactions in Section 4.3.4. Section 4.4 describes our prototype and the results of our evaluation.

4.1 Introduction

ACID transactions provide a simple and powerful abstraction to programmers: transactions appear to complete atomically, one at a time despite executing concurrently. This property greatly simplifies developing and reasoning about multi-threaded applications.

In recent years, we saw a continuing interest in research on transactional systems, e.g., as transactional properties were adopted in unconventional settings such as "NoSQL" systems [18], or as the performance of distributed transactions was improved by leveraging new hardware features [22]. However, this research does not fundamentally improve the performance of transactions when they meet one of their key Achilles heels, contention. This is because, when transactions conflict with one another, they end up executing most of their logic one at a time.

In this chapter, we build on the observation that the lack of semantic information about the transaction leads to a conservative view of what is a conflict, and therefore imposes unnecessary synchronization between transactions. For example, two transactions that increase the number of items in an inventory will be treated as conflicting because they both write to the database tuple containing the total quantity. However, the semantics of those transactions do not imply a conflict, provided that the aggregated effects of both transactions are applied to the database.

To address this shortcoming, we propose lazy state determination (LSD), a novel API for defining transactions that conveys their semantics to the database. The main insight behind LSD is that by exploring the semantics of the transaction, it is possible to increase concurrency while still providing (strict) serializability [55]. This contrasts with most of previous work that explore semantic information to improve transaction processing [8, 54, 57] which focus on maintaining specific application invariants under consistency models weaker than serializability.

One important challenge in our work is how to get useful semantic information without requiring programmers to significantly modify their coding practices. To this end, we realize LSD by having the tx-read operation return a future [7] (an opaque proxy for a value) instead of a concrete value, and materializing futures as late as possible, i.e. only when the transaction commits. To allow transactions to still be expressive with futures without resolving them, we: (a) introduce a new operation, tx-is-true, that allows transactions to specify conditions over futures, and (b) provide operations that allow transactions to specify their updates to the database as lazily-evaluated functions that can depend on futures. Warranties [44] also allow transactions to express conditions that must hold for the transactions to succeed, but is insufficient to help increase concurrency for transactions that perform computations using the values read, or externalize them.

This novel API allows LSD transactions to execute over an abstract database state, and resolve this abstract state as late as possible, thus increasing the chances for safely committing without breaking isolation. To this end, we modified existing optimistic and pessimistic concurrency control protocols to allow for conditional validation. The key idea of this design is to verify that the required conditions still hold when the transaction attempts to commit (in the case of optimistic concurrency control), or to use a condition lock acquired in condition mode for a certain condition *c*, which is only compatible with an acquisition in write mode if the value that will be written respects the condition *c* (in the case of pessimistic concurrency control).

We built and evaluated a prototype transactional key-value store that provides ACID transactions using the LSD interface. LSD transactions achieved up to 5× more throughput with 1.5× less latency than standard transactions under high contention in our experiments with the popular TPC-C benchmark [64].

```
tx-begin
$v \leftarrow$ tx-read($stock$)
if $v \geq qty$
    $v \leftarrow v - qty$
    tx-write($stock$, $v$)
    tx-commit
else
    tx-abort
```

a Traditional interface.

```
tx-begin
$\square \leftarrow$ tx-read($stock$)
if tx-is-true($\{\square \geq qty\}$)
    $\triangle \leftarrow \{\square - qty\}$
    tx-write($stock$, $\triangle$)
    tx-commit
else
    tx-abort
```

b LSD interface.

Figure 4.1: Simplified portion of a TPC-C's New Order-like transaction.

4.2 Overview

A typical database API exposes five operations: (1) tx-begin: starts a new transaction, (2) tx-read(key): returns the value of the database object identified by key, (3) tx-write(key, val): modifies the value of the object identified by key to val, (4) tx-commit: commit the current transaction, and (5) tx-abort: aborts the current transaction.

Conceptually, a transaction is a function f that changes the database from an initial state s_i to a final state s_f, i.e. $f(s_i) = s_f$. In light of this formulation, the tx-read and tx-write API calls allow transactions to specify the final state (tx-write) as a function of the initial state (tx-read).

4.2.1 The pitfalls of the traditional API

Consider the example in Figure 4.1a that depicts a simplified portion of the TPC-C new order transaction [64], which implements the action of buying a certain quantity qty of items. If the item's stock $\big(v \leftarrow$ tx-read($stock$)$\big)$ is enough to fulfill the order $\big(v \geq qty\big)$, the stock value decreases by qty $\big($tx-write($stock$, $v - qty$)$\big)$.

This example illustrates how the tx-read/tx-write interface fails to convey the semantics of the transaction to the database, e.g., the dependencies of the transaction behavior on the values it reads, or how it computes the values it writes. Instead, from the point of view of the database, transactions are a sequence of opaque tx-read/tx-write operations. (Note that this is true regardless of whether transactions execute co-located with the database, as stored procedures, or in a remote client.)

To understand how this can be a limiting factor, consider the situation where the current stock value is 42 ($stock$), and the quantity to order is 1 (qty). When the transaction issues the tx-read($stock$) operation, the database returns the value 42. Since the database does not know what the transaction will do with the returned value, it must be conservative to account for all possible situations, e.g., the transaction only executing some operations depending on the returned value, or using the returned value to perform a computation that returns the value of a subsequent tx-write. As a consequence, 2PL must lock the *stock* object to prevent any other transaction from modifying it and invalidate any branching decision or computed value by the transaction that observed the value 42. Similarly, OCC records the read operation so that the database can check that the *stock*'s value is the same when the transaction attempts to commit; if meanwhile another transaction modifies the *stock*, transactions that observed the now-stale stock value fail to commit.

As this example shows, a central part of enforcing transaction isolation is ensuring that the state that a transaction observes (i.e., the values returned by tx-read operations) remains unchanged throughout its execution. Our key insight is to question whether a transaction *really* needs to observe a specific state during its execution. In other words, in our running example, does the tx-read(*stock*) operation really need to expose a particular state to the transaction before commit (e.g., the value 42)? With the current interface the answer is yes. Otherwise, transactions cannot have conditional branches that depend on the database state, nor perform updates to the state that are a function of that state. Going back to our example, the transaction could not check whether there is enough stock nor compute the new stock value.

4.2.2 Introducing LSD

In this chapter, we overcome these limitations by rethinking the transactional API in order to provide ACID transactions that allow for greater concurrency. The key observation behind LSD is that, in general, transactions do *not* need to observe a concrete state to execute most of their logic. Thus, we propose alternative semantics for the tx-read operation. Specifically, the tx-read operation should *not* expose a specific database state by returning a concrete value, but should instead return a *future* [7].

A future is an object that acts as a proxy for a value that is initially unknown. In our case, a future symbolizes the value of a specific database object. This means that the database *promises* the transaction to resolve the future's value, but does not do it right away. In particular, we want to defer evaluating futures until the transaction attempts to commit (*lazy evaluation* [36]) to maximize concurrency. (Note that the traditional semantics of the tx-read operation is equivalent to returning futures that are immediately resolved.) Returning to our running example, we depict this modification in Figure 4.1b with the future that symbolizes the stock value as □.

The proposed change to the semantics of the tx-read operation has a clear benefit: if a transaction does not observe a specific database state, other transactions can modify it without breaking the isolation guarantees of the first transaction. However, this raises the problem of determining how can a transaction use futures. This can, in turn, be split into two main challenges. The first is how can a transaction perform conditional branching based on futures. The second is how can a transaction compute values that depend on futures. For instance, how can the logic of our example transaction decide whether it can fulfill the order if it does not know the stock value, and how can the transaction compute the new stock value? (A naive approach is to eagerly resolve futures when a transaction requires their value, but this again results in restricting concurrency.)

To solve the first challenge, we observe that a future symbolizes the value of a particular database object. While we would like that a transaction is not able to directly observe the value of a future, we can still ask the database whether a future's value respects a certain condition. For example, the transaction can ask the database whether the stock value is greater than qty, and make a control flow decision depending on the database's answer.

To support this functionality we introduce a new operation, tx-is-true(c), which, given a condition c over one (or more) futures, returns whether the condition holds or not. We show the tx-is-true operation using the $\square \geq qty$ condition in Figure 4.1b. Note that while the tx-is-true operation effectively exposes database state to the transaction, it exposes an abstract state (the stock is greater than qty) rather than a concrete one (the stock is 42), which has the potential to allow for more concurrency, e.g., by allowing concurrent modifications of the stock value as long as it retains a non-negative value after all the modifications.

The second challenge is how can a transaction perform computations using futures. To solve this challenge, we observe that while a transaction cannot perform the actual computation with futures, it can define the necessary computation and let the database perform it when the transaction commits and the futures are resolved to concrete values. For example, the transaction can define that the new stock value is whatever value its future ends up resolving to minus qty.

To support this behavior we change the semantics of the tx-write operation so that, instead of receiving the concrete new value for an object, it receives a function that computes the concrete value when evaluated. This function has the important property that it can depend on the values of any future, since the database can resolve them. Furthermore, tx-write functions are lazily evaluated by the database when the transaction commits, so that the futures that the functions depend on may remain unresolved. In Figure 4.1b, we represent this function as $\{\square - qty\}$, which is the argument of the tx-write operation.

We expect that the proposed changes to the semantics of the tx-read and tx-write operations and the addition of the tx-is-true operation will enable the database to provide ACID transactions with more concurrency, potentially resulting in higher throughput and lower latency, for two main reasons.

The first is that we decrease the time window in which a transaction requires isolation. With the traditional interface, the transaction requires isolation from the moment when it first observes database state (with the traditional tx-read operation) until the transaction attempts to commit. With LSD the transaction only requires isolation during its commit operation if it does not require any specific conditions.

The second reason is that we reduce the set of concurrent transactions that are forced to abort, or wait, when executing concurrently with some transaction to guarantee the

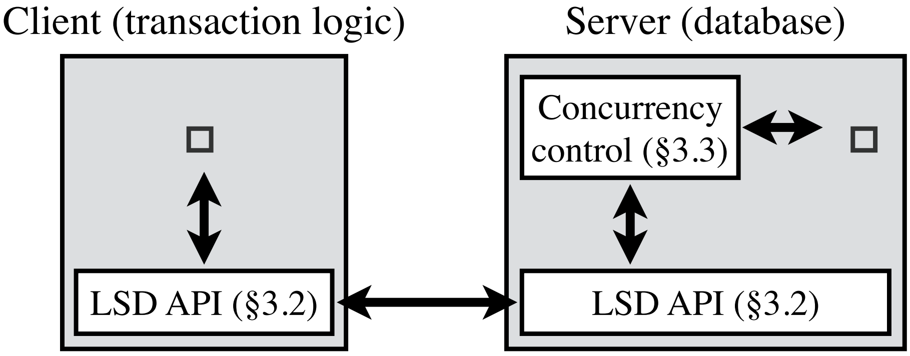


Figure 4.2: Overview of the system's architecture.

required isolation level. Even when a transaction needs to test some condition over database objects, LSD's tx-is-true operation still allows concurrent transactions to modify those objects as long as these modifications do not invalidate the previously asserted conditions. This contrasts with the traditional interface that prevents any modifications, whether they violate such conditions or not. This leads to lower abort rates, or waiting, and hence to a higher amount of useful work performed by the database.

That said, the LSD API is not a panacea. Transactions that must observe a concrete state can not reap LSD's benefits. For example, transactions that externalize values during their execution need to resolve the required futures, falling back to the standard tx-read semantics. However, we believe that a large class of transactions can take advantage of LSD proposed semantics.

4.3 LSD Design

The high-level goal of LSD is to allow databases to provide ACID transactions with higher performance than what can typically be achieved, while minimizing the changes in terms of the way that programmers specify the logic of their transactions.

4.3.1 Design overview

Figure 4.2 shows the main components of our design. Clients execute application code that interacts with the database server via transactions written using the LSD API. Note that these are logical components, meaning that our design does not make assumptions regarding the physical relationship between clients and servers, nor the physical realization of the server. For example, clients can be physically separated from the server or co-located with it (e.g., in a stored procedure), and the database may or may not be partitioned or replicated. Nevertheless, for the rest of this chapter we assume that clients execute transactions and are separated from the server, which is the case in our prototype and evaluation.

Operation	Description
tx-begin	Starts a new transaction.
tx-read(*key*) → □	Returns □, a future for the value of object *key*.
tx-read(△) → □	Evaluates the future △ and returns □, a future for the value of the object that △ evaluates to.
tx-is-true(□) → boolean	Returns whether the condition □ is currently true in the database.
tx-write(*key*, □)	Updates object *key*'s value to the value that □ will evaluate to.
tx-write(△, □)	Updates the value of the object that △ will evaluate to, to the value that □ will evaluate to.
tx-commit → boolean	Attempts to commit the ongoing transaction.
tx-abort	Aborts the ongoing transaction.

Figure 4.3: The LSD interface operations. The symbols □ and △ denote futures.

4.3.2 Interface

Figure 4.3 shows the LSD interface, which allows applications to execute transactions against the database. The tx-begin, tx-commit, and tx-abort operations are the standard operations. They allow an application to start, commit, and abort a transaction, respectively. LSD introduces two changes to the standard interface: new semantics for the tx-read and tx-write operations, and a new tx-is-true operation. We first describe the new tx-read and tx-write operation semantics, then present the tx-is-true operation, and finally address the case when a transaction wants to access an unknown object, i.e., the object identifier is itself a future.

tx-read. The typical semantics of the tx-read operation is to return the current value of the object, which requires the concurrency control protocol to kick in as a result of exposing the database state to transactions. In contrast, LSD's tx-read operation returns a future for the value of a given object, instead of exposing the object's current concrete value. From the application's point of view, this future is an opaque representation of the object's value. However, the database knows how to interpret such future; in particular, it has the possibility to *resolve* the future, i.e., compute the value that the future represents, which is to *actually* read and return the object's value. Thus, informally the contract that LSD provides between the transaction and the database is the following: the transaction should use the future as if it is the actual value, and the database promises to lazily resolve the future such that, when the transaction commits, it is as if it executed with the concrete value instead of the future. The benefit of these semantics is that the concurrency control protocol only needs to intervene when the database resolves a future and *not* when a transaction issues a tx-read operation.

tx-write. The traditional tx-write operation receives both the identifier of the object and its new value. This interface fits well with the traditional tx-read operation since reads return concrete values, so if a transaction wants to modify the value of an object it can read the object, compute the modified value, and write this new value. However, since the LSD tx-read operation returns a future instead of a concrete value, the transaction should be able to modify and write values derived from futures, instead of concrete values. To address this, we have two choices. The first is to resolve the future so that the transaction can perform its modification. This approach goes against LSD's goal, since resolving futures exposes database state to transactions, which in turn requires the concurrency control algorithm to enforce the required isolation. The second choice, which we follow, is *defining but not performing* the computation necessary to modify the value, so that futures may remain unresolved to promote parallelism. To do so, the transaction specifies the computation it needs to do as a function that, when evaluated by the database, computes the new value for the object. For instance in our running example of Figure 4.1b, where a transaction wants to decrease the avaliable stock for a given item, the transaction reads the stock and obtains $\square$ (its future value), and defines the function that decreases the stock ($\square - qty$). This function is also a future: it represents the value that the transaction intends to write to the stock object. For this approach to work, the database needs to know how to evaluate such functions so that, when the transaction commits, the database can install the object's new value. To understand how this can be done, we observe that we can divide this function evaluation into two parts: resolving future reads on which the function depends on, and executing the function's logic. As discussed, the database knows how to resolve future reads. As for executing the function's logic, the idea is that we define this in a way that the database can initially refer to the function without resolving it, but at commit time interpret and execute it. To achieve this, in our prototype, we provide transactions with a library of operations, which can be composed to create functions, e.g., sub($\square$, qty) to decrease the stock in our example.

tx-is-true. So far, we managed to prevent exposing database state to transactions by changing the semantics of the tx-read and tx-write operations. However, transactions may need to decide what to do based on the database state, as exemplified in our running example where the transaction only orders the item if there is enough stock available. As before, we want to avoid resolving the futures required to make the decision of what to do, we introduce the tx-is-true operation, which, given a condition over the database state, returns whether the condition holds or not. This condition is a function that, as discussed for the tx-write operation, can depend on futures. In our running example, the transaction decides to decrease the stock depending on whether there is enough stock available: $\square \geq qty$. (In our prototype, we also provide transactions with operations to create conditions, e.g., gte($\square$, qty), to check whether there is enough stock.)

Note that the tx-is-true operation *does* expose database state to transactions, but this is inevitable if the transaction performs different actions depending on the database state. The merit of the tx-is-true operation is that it exposes *abstract*, instead of *concrete*, state to transactions, which enables the database to maintain isolation while potentially allowing more parallelism. For instance, if the transaction of our running example attempts to purchase a quantity of 4, and the current stock is 42, the tx-is-true operation returns $\top$. Other concurrent transactions may successfully update the stock value and commit without breaking isolation as long as the stock value remains greater or equal to 4. Enforcing the semantics of this operation requires the concurrency control protocol to either: (1) ensure that the result of the tx-is-true operation remains valid until the transaction commits (pessimistic approach, 2PL-style), or (2) abort the transaction when it attempts to commit if the result of the condition no longer holds (optimistic approach, OCC-style). In the next section we discuss how to adapt both 2PL and OCC to support for the tx-is-true operation. We implemented both approaches in our prototype.

Futures as keys. Up until this point, we have not discussed what happens when the transaction attempts to read or write an object whose identifier is itself a future. For reads, this situation is likely to happen when accessing objects via a secondary index. Secondary indexes are seldom kept on keys whose values are updated frequently since they tend to be expensive to modify [63]. Given this observation, we chose to resolve the future immediately when a tx-read operation receives a future as a parameter, in order to know which object is being read. This simplifies reasoning and implementation effort, since the alternative of maintaining "futures of futures" would require a chain of resolves at commit time. As for future identifiers in tx-write operations, we chose to keep them unresolved because transactions may write to objects whose future-keys depend on the database state. This is the case, for example, when assigning unique identifiers to keys from a monotonically increasing counter, which we believe to be a common programming idiom.[1] As such, if we resolve the future identifier immediately, we risk exposing highly-contended database state to transactions, which goes against our design goals. The price we pay for our decision is that, in the general case of distributed transactions, they may require an additional communication round with servers to commit. We discuss this aspect further in Section 4.3.4.

4.3.3 Concurrency control

Now we turn our attention to the impact the LSD API has on concurrency control, and discuss how to adapt two popular and representative concurrency control protocols: OCC [40, 65] and 2PL [11, 24]. The two main elements of the LSD API that drive the adaption are: (1) futures, as the protocol needs to be aware of them to know what to do at commit time, and (2) the tx-is-true operation, which exposes abstract database state to transactions and therefore requires concurrency control.

4.3.3.1 Overview

The high level idea of the adaptation of both OCC and 2PL is to maintain two extra read and write sets, which we call *future read and write sets*, to keep futures unresolved until commit time, and a *condition set* to support the tx-is-true operation and conditions. The LSD-aware OCC and 2PL protocols differ mainly on how they handle the condition set.

[1] This is the case in the popular TPC-C benchmark [64]. The fact that purchase orders have monotonically increasing identifiers not only guarantees uniqueness, but also serves to identify and compare the recency of each order.

OCC verifies that the conditions still hold at commit time while 2PL ensures that concurrent transactions that write values that invalidate active conditions cannot commit while such conditions are active. Otherwise, the new protocols follow the essence of the original protocols.

Figures 4.4 and 4.6 show the LSD-aware OCC and 2PL protocols, respectively. The behavior of the tx-begin, tx-read(*key*), and tx-write operations is protocol-agnostic so we start by describing these before detailing the protocols for OCC and 2PL in Section 4.3.3.2 and Section 4.3.3.3, respectively.

tx-begin. Initializes the read/write sets, future read/write set, and condition set (*rset*, *wset*, *frset*, *fwset*, and *cset*, respectively.)

tx-read(*key*). Creates a future-value for *key*'s value (□), add it to the future read set, and returns it. (This is a local operation).

tx-write(*key*,□). Buffers □, the future-value for *key*, in the write set.

tx-write(△,□). Buffers □, the future-value to assign the future-key △, in the future write set.

4.3.3.2 Optimistic concurrency control (OCC)

In a nutshell, OCC works as follows. Each database object is associated with a version. Reads record the object identity and the observed version in the *read set*. Writes are buffered in the *write set* until the transaction attempts to commit, instead of modifying the database immediately. Then, when a transaction attempts to commit, it atomically verifies if every object in the read set is unchanged, i.e., if it is still in the same version that was read, and, if so, all buffered updates are applied, and the respective version numbers are incremented. This atomic test and change is implemented in three steps: (1) lock the write set, (2) validate the read set, and (3) perform the pending writes, if the validation was successful, and release the acquired locks.

Next, we describe the adaptations required for the remaining operations, as depicted in Figure 4.4.

tx-read(△). Resolves the future-key △, i.e., compute its concrete value $value$ and add the observed version to the read set, and then tx-read($value$). (Returning a future.)

tx-is-true(□). Observes the current value of each key present in the condition □, i.e., each future-value over which the condition is defined, resolve □ using the observed values, add the result to the condition set, and returns it.

tx-commit. As we discussed, the commit protocol executes in three steps. First, we lock the write set and the future-write set. However, the latter initially has its keys unresolved. To resolve and then lock them, we first need to resolve the future read set because it contains the future-values of tx-read operations that were delayed, and future-keys and future-values in the regular and future write sets are likely to depend on the future read set. (e.g., a transaction reads *key* and gets future-value □, which it then uses to create a future $△ = f(□)$, according to some function f, that the transaction uses as a

```
tx-begin
    $rset \leftarrow \emptyset$
    $wset \leftarrow \emptyset$
    $frset \leftarrow \emptyset$
    $fwset \leftarrow \emptyset$
    $cset \leftarrow \emptyset$
tx-read($key$)
    $\Box \leftarrow$ future($key$)
    $frset \leftarrow frset \cup \{\Box\}$
    return $\Box$
tx-read($\triangle$)
    $key \leftarrow$ key($\triangle$)
    $\langle value, version \rangle \leftarrow$ get($key$)
    $rset \leftarrow rset \cup \{\langle key, version \rangle\}$
    return tx-read($value$)
tx-write($key$, $\Box$)
    $wset \leftarrow wset \cup \{\langle key, \Box \rangle\}$
tx-write($\triangle$, $\Box$)
    $fwset \leftarrow fwset \cup \{\langle \triangle, \Box \rangle\}$
tx-is-true($\Box$)
    $rvalues \leftarrow \emptyset$
    foreach $key \in$ keys($\Box$) do
        $rvalues \leftarrow rvalues \cup \{\langle key,$ get($key$)$\rangle\}$
    $result \leftarrow$ resolve($\Box$, $rvalues$)
    $cset \leftarrow cset \cup \{\langle \Box, result \rangle\}$
    return $result$
```

```
tx-commit
    $rvalues \leftarrow \emptyset$
    $result \leftarrow \top$
    foreach $\langle key, - \rangle \in wset$ do
        lock($key$)
    foreach $\Box \in frset$ do
        $key \leftarrow$ key($\Box$)
        lock($key$)
        $rvalues \leftarrow rvalues \cup \{\langle key,$ get($key$)$\rangle\}$
    foreach $\langle \triangle, \Box \rangle \in fwset$ do
        $key \leftarrow$ resolve($\triangle$, $rvalues$)
        lock($key$)
        $wset \leftarrow wset \cup \{\langle key, \Box \rangle\}$
    foreach $\langle key, version \rangle \in rset$ do
        if $version \neq$ version($key$) then
            $result \leftarrow \bot$
    foreach $\langle \Box, expected \rangle \in cset$ do
        $value \leftarrow$ resolve($\Box$, $rvalues$)
        if $value \neq expected$ then
            $result \leftarrow \bot$
    if $result = \top$ then
        foreach $\langle key, \Box \rangle \in wset$ do
            $value \leftarrow$ resolve($\Box$, $rvalues$)
            $version \leftarrow$ next-version($key$)
            put($key$, $value$, $version$)
    foreach $key \in wset \cup$ keys($frset$) do
        unlock($key$)
    return $\langle result, rvalues \rangle$
```

Figure 4.4: LSD-aware OCC protocol.

future-key — tx-write(△,...) — and/or future-value — tx-write(...,△).) To guarantee that we resolve the future read set consistently, we first lock the respective keys.

In the second step, we validate the read set. In addition to the read set, transactions also observe database state via conditions and the tx-is-true operation, so we also validate each condition in the condition set using the values obtained from the future read set. Finally, in the third step, we resolve the buffered future-values, perform the writes, and release acquired locks.

To illustrate these steps, we will simulate the execution of our running example of Figure 4.1b. First, the transaction issues the tx-read operation for the item's stock. This operation is local to the client, since it merely creates the future □ and returns it. Then the transaction attempts to purchase qty amount of items if there is enough stock. Let us assume that $qty = 10$. Since the transaction does not know the concrete value of the item's stock, it uses the tx-is-true operation to check whether there are at least 10 items available. Assume that, in this example execution, the transaction is operating on a database state where there are at least 10 items in stock. Then, in order to maintain

	R	W	R(p)	W$\big(v : c(v)\big)$	W$\big(v : \neg c(v)\big)$
R	✓	–	✓	–	–
W	–	–	–	–	–
R(p)	✓	–	✓	✓	–
W$\big(v : c(v)\big)$	–	–	✓	–	–
W$\big(v : \neg c(v)\big)$	–	–	–	–	–

Figure 4.5: 2PL's lock compatibility matrix. The gray cells represent the standard 2PL matrix, and LSD introduces the remaining cells. A ✓ means that acquiring the lock in row mode succeeds when the lock is in column mode.

isolation, this condition must also hold when the transaction attempts to commit, and thus the transaction records the condition and its result in the condition set for commit-time validation. Finally, the transaction defines the necessary computation to update the stock value with the future △, issues the tx-write with it, and attempts to commit. The tx-commit operation will then atomically resolve the stock value □ to, for example, 42, verify that $42 \geq 10$, and compute the new stock value △ to be $42 - 10 = 32$. Note that, when using standard OCC, any concurrent write to the stock value causes the transaction to abort. With LSD-aware OCC, instead, the transaction only aborts if between the time the tx-is-true and tx-commit operations are issued the stock value changes to a value below 10.

Possible optimization. Since the tx-is-true operations are validated at commit time to ensure isolation, it is possible to optimistically assume a specific result for an tx-is-true operation *without* communicating with the database. Whether this behavior yields better performance or not depends on the success rate of the assumption: if the assumption is correct we save one communication round with the database, but if it is not, the transaction aborts, perhaps needlessly, and upon retry performs the tx-is-true operation normally. We evaluate this optimization in Section 4.4.

4.3.3.3 2-phase locking (2PL)

2PL follows a rational opposite to OCC: instead of assuming that conflicts seldom happen, 2PL immediately acquires a lock when a transaction accesses an object to prevent conflicting transactions from proceeding in parallel and breaking isolation.

The central idea of the adaptation of 2PL to LSD's tx-is-true operation is the novel concept of a *condition lock*, which is an extension of a *read-write lock*. To understand the semantics of condition locks, we first recall that read-write locks can be acquired in either read or write mode (R or W). The semantics of read-write locks are then given by their compatibility matrix shown in gray in Figure 4.5. This shows that multiple readers, i.e., read-mode acquires, can proceed simultaneously, but writers are serialized. Condition locks, in turn, have two additional modes: *read condition* and *write value*. The read condition mode, R(c), essentially associates a condition c with the lock, signaling

```
tx-begin
    $rset \leftarrow \emptyset$
    $wset \leftarrow \emptyset$
    $frset \leftarrow \emptyset$
    $fwset \leftarrow \emptyset$
    $cset \leftarrow \emptyset$
tx-read($key$)
    $\square \leftarrow$ future($key$)
    $frset \leftarrow frset \cup \{\square\}$
    return $\square$
tx-read($\triangle$)
    $key \leftarrow$ key($\triangle$)
    lock($key$)
    $value \leftarrow$ get($key$)
    $rset \leftarrow rset \cup \{key\}$
    return tx-read($value$)
tx-write($key$, $\square$)
    $wset \leftarrow wset \cup \{\langle key, \square\rangle\}$
tx-write($\triangle$, $\square$)
    $fwset \leftarrow fwset \cup \{\langle \triangle, \square\rangle\}$
tx-is-true($\square$)
    $rvalues \leftarrow \emptyset$
    foreach $key \in$ keys($\square$) do
        lock($key$)
        $rvalues \leftarrow rvalues \cup \{$get($key$)$\}$
    $result \leftarrow$ resolve($\square$, $rvalues$)
    foreach $key \in$ keys($\square$) do
        add-condition($key$, $\langle\square, result\rangle$)
        unlock($key$)
    $cset \leftarrow cset \cup \{\square\}$
    return $result$
```

```
tx-commit
    $rvalues \leftarrow \emptyset$
    foreach $\square \in frset$ do
        $key \leftarrow$ key($\square$)
        lock($key$)
        $rvalues \leftarrow rvalues \cup \{\langle key,$ get($key$)$\rangle\}$
    foreach $\square \in cset$ do
        foreach $key \in$ keys($\square$) do
            rem-condition($key$, $\square$)
    $set \leftarrow \emptyset$
    foreach $\langle\triangle, \square\rangle \in fwset$ do
        $key \leftarrow$ resolve($\triangle$, $rvalues$)
        $set \leftarrow set \cup \{\langle key, \square\rangle\}$
    $writes \leftarrow \emptyset$
    foreach $\langle key, \square\rangle \in wset \cup set$ do
        $value \leftarrow$ resolve($\square$, $rvalues$)
        $writes \leftarrow writes \cup \{\langle key, value\rangle\}$
    foreach $\langle key, value\rangle \in writes$ do
        lock-compatible($key$, $writes$)
        put($key$, $value$)
    foreach $key \in writes \cup rset \cup$ keys($frset$) do
        unlock($key$)
    return $\langle\top, rvalues\rangle$
```

Figure 4.6: LSD-aware 2PL protocol.

that a transaction has observed a value that respects the condition c. Other transactions can still successfully update a read condition-locked object by acquiring the lock in write value mode. The write value mode, $\mathrm{W}(v)$, is aware of the value v that the transaction intends to assign to the object. If the lock is in read condition mode and the value v respects all the conditions that the lock holds, the write mode acquire succeeds. Otherwise it blocks as usual. Note that the read condition mode is a generalization of the read mode: the latter is smilar to the former with a condition that always returns false regardless of the value other transactions intend to write.

Next, we describe the adaptations required for the remaining operations, which are also summarized in Figure 4.6.

tx-read($\triangle$). Resolves the future-key $\triangle$, i.e., compute its concrete value $value$ by locking key, reading its value, and then tx-read($value$). (Returning a future.)

tx-is-true($\square$). Atomically observes the current value of each key present in the condition $\square$ by locking all keys. Resolve the condition $\square$ using the observed values, and downgrade the acquired locks to read condition mode using □ and its result.

tx-commit. Resolves the future read set by locking and performing the delayed reads. Remove the conditions installed via the tx-is-true operations since we already resolved the future read set. Resolve all future-keys in the future write set, and all future-values in the future and concrete write set. Given that we now know the transaction's full write set, acquire the locks in write value mode, perform the writes, and release the acquired locks.

Again, to better understand these steps, we will go through the steps of the execution of our running example of Figure 4.1b. The transaction reads the item's stock, which is an operation local to the client. Then the transaction attempts to purchase qty amount of items if there is enough stock. Let us assume that $qty = 10$. The transaction uses the tx-is-true operation to check whether there are at least 10 items available. Again, assuming that this is the case, to maintain isolation this must be also true when the transaction commits. To ensure this, a condition lock is acquired, in read condition mode, on the stock stating that its value must remain greater or equal to 10. The transaction proceeds to define the necessary computation to update the stock value with the future △, issues the tx-write with it, and attempts to commit. The tx-commit operation will then atomically resolve the stock value □ to, for example, 42, remove the condition $\square \geq qty$ from the stock lock, and compute the new stock value △ to be $42 - 10 = 32$. Then the transaction acquires the stock's lock in write value mode with 32, blocking only if there is any concurrent reader that installed a condition c such that $\neg c(32)$. (With standard 2PL *any* concurrent reader would cause the transaction to block.) Finally, the transaction modifies the stock to 32 and releases the locks.

4.3.3.4 Multi-future conditions

OCC and 2PL fundamentally differ on how they deal with the validity of conditions. OCC does not ensure that a condition asserted via the tx-is-true operation remains valid. This is because write transactions are not aware of those conditions and can freely violate the conditions when they commit. As such, it is up to a transaction that asserts a condition to validate it when the transaction attempt to commit to ensure isolation, i.e., the burden of dealing with conditions is on the readers. In constrast, 2PL ensures that an asserted condition remains valid until the asserting transaction commits, as acquring a condition lock in write value mode will block if the value to be written violates any existing asserted condition, i.e., the burden of dealing with conditions is on the writers.

Dealing with conditions on the writer's side is more complex than on the reader's, and this complexity is exacerbated in the presence of conditions that encompass more than one future (multi-future conditions). For example, consider two keys x and y, with values 2 and 1, respectively, read by some transaction t_1 as futures □ and △. t_1 then executes tx-is-true($\{\square > \triangle\}$), which returns $\top$ (because $2 > 1$). Then assume that, concurrently, another transaction t_2 attempts to write 1 to x. For t_2 to acquire x's condition lock in write value mode with value 1 and commit, the procedure to acquire the condition lock in write value mode (lock-compatible in Figure 4.6) can only grant the lock to t_1 if $1 > \triangle$ remains valid. Thus, the locking procedure must resolve $\triangle$ to check the concrete validity of $1 > 1$. To do so, there are two possibilities. If t_2 also reads y, then it has acquired a read lock on y so it can safely resolve $\triangle$. If not, the lock procedure needs to resolve $\triangle$ in a way that still ensures transactional isolation, e.g., by acquiring a read lock on y on behalf of t_2.

Given the experience in the implementation of our prototype, we argue that the tx-is-true operation is simpler to implement, and understand, using an optimistic approach. Additionally, the experimental evaluation (Section 4.4) using our prototype shows that the LSD-aware OCC protocol performs better than the LSD-aware 2PL protocol, so we conclude that future implementations of LSD should use OCC in most cases.

4.3.4 Distributed transactions

So far we have discussed how to adapt both OCC and 2PL to exploit LSD in the context of a single server. However, transactions may be distributed, i.e., span multiple servers, if the database is partitioned. We now briefly sketch how to adapt 2-phase commit (2PC) [11], the most widely used distributed commit protocol, to support LSD.

LSD introduces the future read and write sets, and condition set. The future write set is of particular importance, since it depends on the future read set. This means that, in general, transactions that have a non-empty future write set require an additional round of communication during 2PC's prepare phase. Each participant resolves, and returns, its portion of the future read set in the regular communication round of the prepare phase. Armed with the resolved future read set the coordinator can resolve the future write set and send it to the required participants.

It is possible to circumvent the need for the additional communication round in the prepare phase and send the future write set immediately in the first round if, for every entry in the future write set: (1) we can identify its future-key's partition without resolving it, and (2) (all) the future(s) on which the future-key depends is (are) from the same partition it belongs to. In our experiments we evaluate both cases: when LSD incurs in an additional communication round in 2PC, and when it does not.

4.4 Evaluation

We implemented a partitioned, transactional, key-value store prototype, including all of the previously described design with the exception of multi-future conditions. Each partition is implemented as a Thrift [5] non-blocking server, and data is stored in disk using RocksDB [25]. Clients can execute transactions using the typical API (tx-begin, tx-read, tx-write, tx-commit, and tx-abort operations) or LSD's API which features our proposed tx-read, tx-write, and tx-is-true operations. We implemented both classical OCC and 2PL, and also both their LSD-aware variants for LSD transactions. Distributed transactions commit using 2PC. We resolve deadlocks that may arise in 2PL or 2PC using the wound-wait strategy.

We conducted an experimental evaluation of our LSD prototype on a private gigabit ethernet cluster. Each server runs on a machine with a 2Ghz Intel Xeon E5-2620 processor, 32GB of RAM, and a 7200 RPM hard drive. Clients run on the various remaining machines with AMD and Intel processors, and communicate with the servers using Thrift RPCs.

Each data point reports the average of 5 runs. Our evaluation seeks to answer the following questions:

- Does LSD improve the performance of realistic applications under contention? (Section 4.4.1.1)

- What is LSD's overhead when contention is low? (Section 4.4.1.2)

- How do LSD's benefits vary across various deployment scenarios, such as with a single database, or with a partitioned database and distributed transactions? (Section 4.4.1)

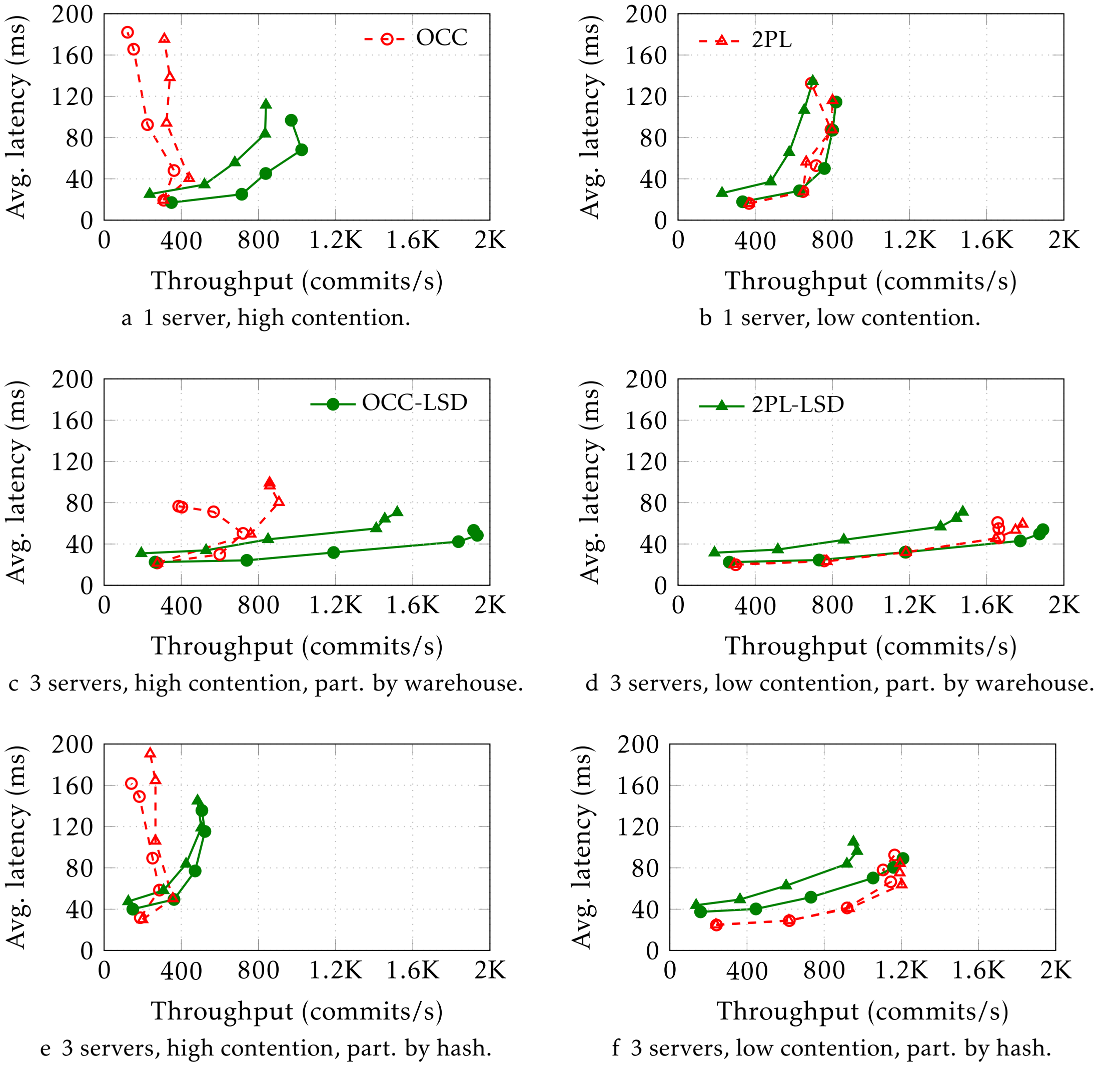
- What is the impact of an increasing amount of contention with and without conditions? (Section 4.4.2)

4.4.1 Realistic application: TPC-C

We used the popular TPC-C benchmark [64] to assess LSD's ability to improve performance of realistic applications under contention, as well as its overhead, on different deployment scenarios. LSD was particularly helpful for the two core transactions of the workload: Payment and New Order. For example, both make use of write functions to modify client balance and stock values, and the latter also uses conditions.

We experimented with TPC-C under three different deployments: (a) a centralized database, (b) a partitioned database using an application-specific partitioning policy, and (c) a partitioned database using an application-agnostic partitioning policy. We executed TPC-C with a high and low contention workload in each deployment.

Setup. We setup each deployment as follows. The centralized database (a) uses a single server that stores the entire data. The database partitioned using an application-specific policy (b) uses 3 servers. The data associated with a particular warehouse is stored within a single server. The remaining data, such as item information, is partitioned across all servers via hashing. Finally, the database partitioned using an application-agnostic policy (c) also uses 3 servers. Data is partitioned across all servers via hashing.



a 1 server, high contention.

b 1 server, low contention.

c 3 servers, high contention, part. by warehouse.

d 3 servers, low contention, part. by warehouse.

e 3 servers, high contention, part. by hash.

f 3 servers, low contention, part. by hash.

Figure 4.7: Performance of TPC-C on a workload using: 1 server with high (a) and low (b) contention; 3 servers with partitioning by warehouse (b,e); and 3 servers with partitioning by hash (c,f).

4.4.1.1 High contention

In TPC-C, the level of contention is proportional to the number of warehouses, so we loaded the database with the minimum number of warehouses applicable to each deployment (as detailed below) and then executed TPC-C with an increasing number of clients. Figures 4.7a, 4.7c, and 4.7e, compare the throughput, measured in committed transactions per second (x axis), and the corresponding average transaction execution latency, measured in in milliseconds (y axis), of OCC and 2PL with and without LSD.

Centralized deployment (Figure 4.7a). We began by loading the database with 1 warehouse. The LSD-aware OCC variant achieved a peak throughput of ≈ 1K committed transactions per second with an average latency of ≈ 70 ms, which amounts to ≈ 6.5× higher throughput and ≈ 2.5× lower latency than standard OCC under the same load. The

LSD-aware 2PL variant achieved a peak throughput of ≈ 850 committed transactions per second with an average latency of ≈ 80 ms, which amounts to ≈ 2.5× higher throughput and ≈ 1.5× lower latency than standard 2PL under the same load.

Partitioned deployment using application-specific policy (Figure 4.7c). For this deployment we loaded the database with 3 warehouses. Data was partitioned across the servers by warehouse, i.e., each server hosts a single warehouse. This scenario allows for the presence of distributed transactions. Distributed LSD transactions commit using the regular 2PC protocol, i.e., without incurring in the additional communication rounds discussed in Section 4.3.4, thanks to the application-specific partitioning policy. The LSD-aware OCC variant achieved a peak throughput of ≈ 2K committed transactions per second with an average latency of ≈ 50 ms, which amounts to ≈ 5× higher throughput and ≈ 1.5× lower latency than standard OCC under the same load. The LSD-aware 2PL variant achieved a peak throughput of ≈ 1.5K committed transactions per second with an average latency of ≈ 60 ms, which amounts to ≈ 1.5× higher throughput and ≈ 1.3× lower latency than standard 2PL under the same load.

Partitioned deployment using application-agnostic policy (Figure 4.7e). For this experiment, we loaded the database with a single warehouse, and all data is partitioned across the servers using hashing. By using an application-agnostic partitioning policy, such as hashing, distributed LSD transactions may need an additional communication round to commit using 2PC. This is the case for the New-Order transaction, which comprises almost half of the workload. Despite the additional communication round, the LSD-aware OCC variant achieved a peak throughput of ≈ 500 committed transactions per second with an average latency of ≈ 120 ms, which amounts to ≈ 2.8× higher throughput and ≈ 1.3× lower latency than standard OCC under the same load. The LSD-aware 2PL variant achieved a peak throughput of ≈ 500 committed transactions per second with an average latency of ≈ 120 ms, which amounts to ≈ 1.8× higher throughput and ≈ 1.3× lower latency than standard 2PL under the same load.

Discussion. This workload highlights the benefits of LSD. For example, under the standard interface semantics, any two concurrent New-Order transactions conflict if: (a) they operate on the same district (conflicting accesses to the district's order identifier counter), or (b) they order the same item (conflicting accesses to the item's stock). Under OCC only one of the concurrent transactions commits and the other aborts. Under 2PL one of the transactions queues behind the other when it attempts to acquire the lock held by the other. In both cases one of the transactions effectively prevents the other from executing, leading to an effective serialization of their execution. With LSD, New-Order transactions delay their accesses to the district's order identifier counter until commit time, so these accesses do *not* result in aborts under OCC, nor queueing during transaction execution under 2PL. Furthermore, any two New-Order transactions that order the same item only conflict if both attempt to buy the entire remaining stock. LSD's benefits translate in practice to higher throughput and lower latency under contention due to less aborts (resp. blocks) under OCC (resp. 2PL). For example, in the data point where LSD transactions achieve their peak throughput on Figure 4.7a, ≈ 92% of OCC transactions abort, whereas this number drops to ≈ 8% with the LSD-aware variant.

It is worth noting that our LSD-aware 2PL implementation incurs in higher overhead than its OCC counterpart. While there still may be room for optimization of our prototype, the LSD-aware 2PL has fundamentally more overhead than its OCC counterpart because condition locks are a more complex technique than condition validation.

4.4.1.2 Low contention

In the previous section, we evaluated LSD using a TPC-C workload with high contention, which is the type of workload that LSD can benefit from. In this section we describe our evaluation of LSD in the opposite scenario: a TPC-C workload with low contention. Specifically, we increased the number of warehouses in the workload from 1 to 32.

In both the centralized (Figure 4.7b) and partitioned deployment using the application-specific policy (Figure 4.7d) we observe that the LSD-aware OCC variant incurred in marginal overhead. In the partitioned deployment using the application-agnostic policy (Figure 4.7f), the overhead becomes more pronounced (≈ 1.25–1.5×) due to the additional communication round needed to commit some distributed transactions. However, at high load the LSD-aware variant managed to achieve similar to better performance. In contrast, the LSD-aware 2PL consistently exhibits worse performance than either concurrency control protocol using the standard interface.

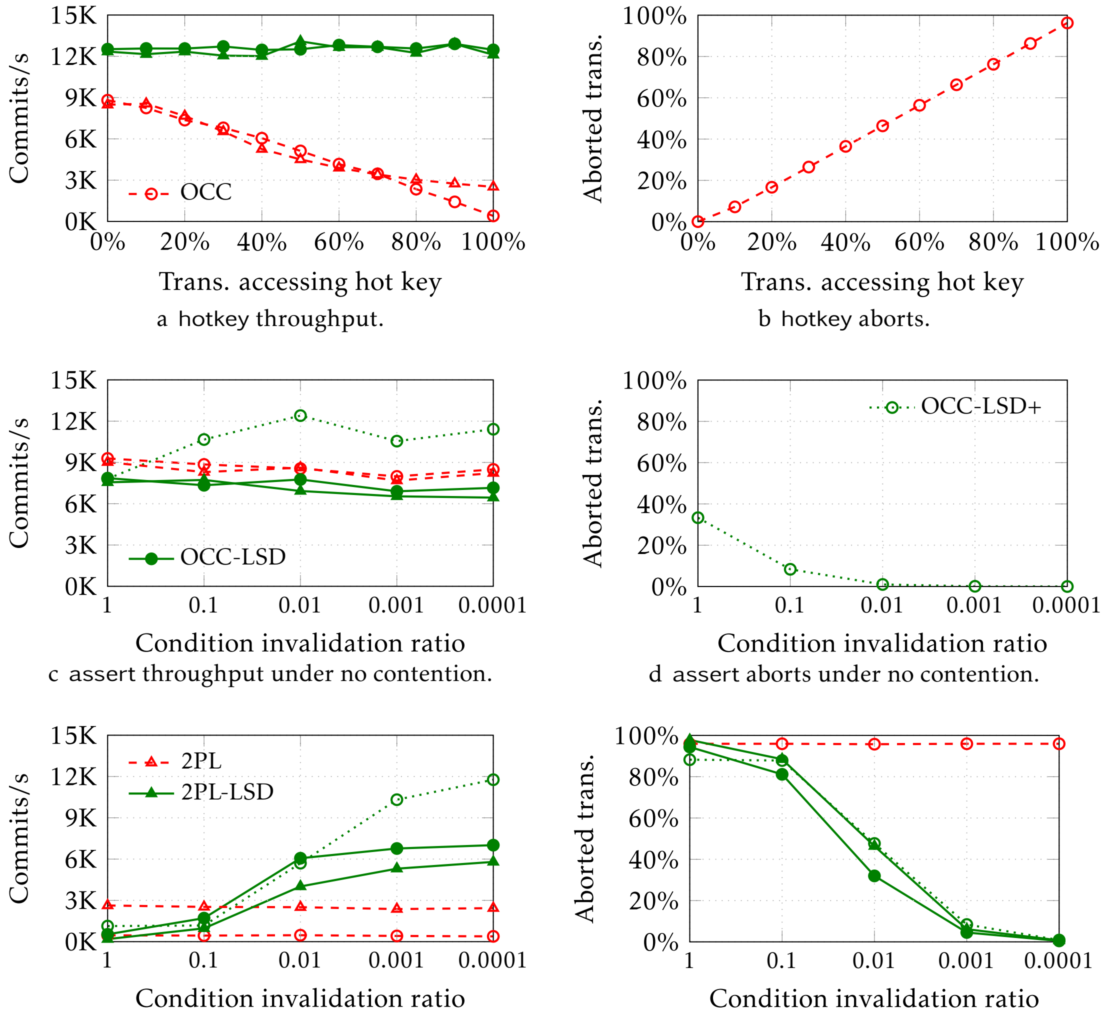
We conclude that the LSD-aware OCC protocol is not only the best of the LSD variants, but also the best solution when either using a single database or a partitioned database with a partitioning scheme that allows for committing distributed transactions without incurring in additional communication rounds. Even with additional communication rounds, LSD is able to reap better performance under contention, while still providing competitive performance when contention is low.

4.4.2 Microbenchmarks

In this section we report on microbenchmark results that show the effect of specific workload characteristics on LSD.

Contention without conditions. We start by analyzing the effect of contending read-modify-write operations. To do so, we loaded the database with as many private counters as there were clients, and a single shared counter—the “hot” counter. Transactions consisted of an increment of either the hot counter, according to some probability p, or the respective private counter, with probability $1 - p$. We executed the microbenchmark for various values of p, ranging from 0% (no contention) to 100% (all transactions contend).

Figure 4.8a plots the measured throughput as a function of the parametrized contention. The LSD-aware protocols are not affected by the parameter because the increments are delayed until commit time, whereas the throughput of the OCC and 2PL protocols decreases when contention increases, as expected, due to aborts in OCC (Figure 4.8b),



a hotkey throughput.

b hotkey aborts.

c assert throughput under no contention.

d assert aborts under no contention.

e assert throughput under high contention.

f assert aborts under high contention.

Figure 4.8: Throughput and aborts on the hotkey (a,d) and assert microbenchmarks with no (b,e) and high (c,f) contention.

and transactions blocking when attempting to read the value of the hot counter in 2PL. At 100% contention, LSD's throughput is ≈ 5× higher that 2PL and ≈ 30× more than OCC.

Note that even when every transaction only increments its own private counter, the LSD-aware variants still perform better than their standard counterparts: this is due to the fact that the LSD's tx-read operation does not communicate with the database (it creates the respective future locally). Hence, LSD transactions incur in less communication rounds than standard transactions, which translated into an ≈ 1.3× increase in throughput.

Contention with conditions. We now analyze the effect of contention in the presence of conditions asserted with the tx-is-true operation. Like in the previous microbenchmark, we loaded the database with a set of private counters and a single hot counter. These counters are initialized with a parametrized value n, and a parametrized percentage of transactions access the hot counter while the remaining access their private counter. The logic of the transactions consisted of decrementing the value of the counter if it remained greater than zero, or restoring the its initial value otherwise. Unlike the previous experiment, in this one we could control the contention that LSD transactions experienced on the condition: the smaller the initial value of the counters, the higher the contention, i.e., the condition "the counter remains greater than zero" changes at a rate of $\frac{1}{n}$, where n is the parameterized initial value for the counters.

Figures 4.8c, 4.8e, 4.8d, and 4.8f, depict the throughput and abort percentage of each protocol. For a scenario with no contention for either LSD or the standard interface, i.e., each transaction only accesses its private counter, the LSD variants incur in an overhead of ≈ 1.1–$1.25\times$ when compared to their standard counterparts (Figure 4.8c). This overhead comes from the additional work performed by the tx-is-true operation, which is not extracting additional parallelism in this experiment because there is no contention. We also plot a version of the LSD-aware OCC (OCC-LSD+) that assumes the counter's value remains greater than zero after the decrement, i.e. it speculates the outcome of the tx-is-true operation without contacting the database, as discussed in Section 4.3.3.2. The effectiveness of the lsd+ variant depends on the success of its speculation. As expected, the results in Figure 4.8c show that the throughput of the lsd+ variant increased when we decreased the condition invalidation ratio, increasing throughput up to $\approx 1.3\times$ that of the standard protocols. The throughput increases because the number of aborts due to failed speculation decreases, as shown in Figure 4.8d. Only the lsd+ variant aborts in this experiment because each transaction accesses its own private counter.

Next, we examined the situation where all transactions access the hot counter. This is the worse case scenario for the standard transactions, whereas LSD transactions can still extract parallelism if the concurrent modifications to the counter do not keep invalidating the condition. Figure 4.8e reports the observed throughput as a function of the condition invalidation ratio. The performance of standard transactions is unaffected by the condition invalidation ratio because standard transactions only deal with concrete values when accessing the counter, so all concurrent transactions conflict: OCC suffers from a high percentage of aborts (Figure 4.8f) while 2PL suffers from a "queueing" effect when acquiring the lock in the tx-read operation. Note that in this experiment the results for 2PL are optimal somewhat inflated, because we disabled deadlock prevention for 2PL since transactions only access a single key. With LSD, on the other hand, throughput increased as there was more available parallelism to exploit, i.e., updates to the counter that would not make its value fall below 1. In particular, as the abort percentage decreased (Figure 4.8f), the LSD-aware variant of OCC (resp. 2PL) achieved up to $\approx 17\times$ (resp. $\approx 2\times$) more throughput than its standard counterpart (Figure 4.8e). The lsd+ variant was able to further boost the throughput gains to $\approx 30\times$ the performance of OCC.

Chapter 5

Concurrent state machine replication using Pot+LSD

In this chapter we combine Pot (chapter 3) and LSD (chapter 4) to realize the vision of this dissertation: executing requests concurrently in state machine replication. We evaluate our implementation of a prototype using the TPC-C benchmark. [64]

The chapter is organized as follows. Section 5.1 is a primer on Pot and LSD. Section 5.2 discusses how LSD is combined with Pot. Section 5.3 describes our prototype and the results of its experimental evaluation.

5.1 Background

5.1.1 Pot

This dissertation's research proposes to extract concurrency from the state machine replication execution phase by executing requests as transactions. We argue that this approach is attractive because it keeps the programming model unchanged: a simple, sequential, programming model where the developer does not need to reason about the complex subtleties of concurrency. The system deals with concurrency transparently and automatically.

To realize our vision, however, it is not enough to simply execute requests as transactions. The problem is that traditional concurrency control protocols that implement serializability, [55] such as two phase locking [11] or optimistic concurrency control, [40] perform two tasks simultaneously while transactions execute: (a) they compute the transaction serialization order (ordering), and (b) control the concurrent execution of transactions to respect that serialization order (concurrency control). Since ordering is intertwined with concurrency control, the final transaction serialization order depends on the nondeterministic interleaving that occurs at runtime between transactions and thus can, and likely will, vary across replicas. We refer to this execution model as *traditional transactions*.

We address the problem of different serialization orders across replicas using *preordered transactions*. With preordered transactions the serialization order is independent of the interleaving that may occur between transactions because, unlike traditional transactions, preordered transactions already have a place in the serialization order before they are executed. Conceptually, preordered transactions have a two-phase execution model: (1) the ordering phase which defines every transactions' place in the serialization order, and (2) the execution phase where transactions execute concurrently in such a way that the outcome is equivalent to their sequential execution in the predefined order.

State machine replication's agreement phase maps directly to the transactions' ordering phase: all replicas agree on a common serialization order. What is left is for the concurrency control protocol in the execution phase to respect the predefined order. Traditional concurrency control protocols cannot be used in this context because they implement both ordering and concurrency control, so we propose a new concurrency control protocol that can.

Our new concurrency control, which we call Pot (short for **p**re**o**rdered **t**ransactions), only performs concurrency control because the task of ordering transactions is offloaded to the state machine replication's agreement phase. Pot takes the serialization order defined in the agreement phase and enforces that order.

We design Pot by modifying optimistic concurrency control, which works as follows. An optimistic transaction consists of one, or more, speculative executions. A speculative execution is divided into three phases: (1) the read phase, (2) the validation phase, and (3) the write phase. The read phase records the objects read by the transaction in the transaction's read set. In the read phase, write operations do not modify the shared state; instead the transaction defers its updates and logs them in its write set. Therefore locations that are both read and modified occur in both the read and the write set. After the read phase, the transaction undergoes a validation phase where it checks whether any concurrently committed transaction's updates overlap with its read set. If so the transaction is aborted to respect serializability, and can be retried; otherwise it proceeds to the next phase. Finally, the transaction enters the write phase where it atomically updates all objects in its write set with the values buffered during the read phase.

The protocol just described provides the illusion that transactions execute one at a time. However, the order in which transactions appear to execute is not deterministic because it depends on the interleaving between transactions' operations that will occur at runtime. To adhere to the serial order predefined in the ordering phase, we make two key observations: (a) optimistic transactions only modify shared state during their write phase, and (b) each transactions' place in the serialization order depends on the relative order in which each transaction (atomically) performs its validation and write phase. Therefore, we restrict transactions to execute their validation and write phases in

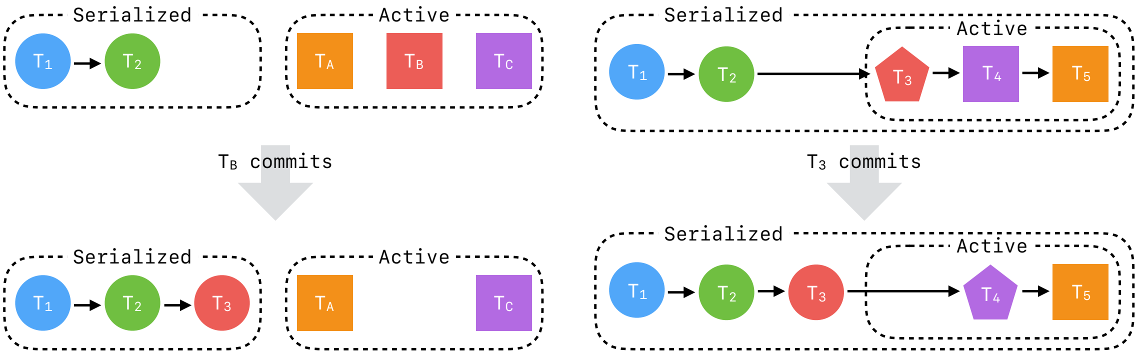


Figure 5.1: Traditional transactions (left) vs. preordered transactions (right). The serialization order of traditional transactions is unpredictable because active traditional transactions can commit in any order, which is likely to be different across replicas. In contrast, the serialization order of preordered transactions is predefined, and therefore deterministic across replicas.

the order defined in the state machine replication's agreement phase, which guarantees that the outcome is equivalent to the sequential execution in the order defined in the agreement phase. We call this technique *ordered commits*.

While ordered commits on its own is enough to guarantee that the agreed upon transaction order is respected, the protocol employs a set of techniques to guarantee correctness, such as read and write sets, read set validation and deferred updates. With optimistic concurrency control all transactions are executed using the aforementioned techniques because any transaction may become the next transaction in the serialization order, which is being defined as transactions execute. Using such techniques imposes additional overhead when compared with an execution without any concurrency control. However, in Pot the serialization order is predefined. Since Pot restricts the order in which transactions commit, they may now have to wait for their turn to commit, leading to a loss of parallelism. To mitigate this loss of parallelism, we make the key observation that at any moment there is always a single transaction, which we refer to as *fast*, which is the next transaction that is allowed to commit. We exploit the fact that the fast transaction is the next transaction allowed to commit to execute it without most concurrency control overheads. Hence, we distinguish between two types of transactions: *fast* and *speculative*. We call this technique *transaction modes*.

A fast transaction is the only active transaction whose predecessors are all completed. A fast transaction is the next, and only, transaction allowed to commit. It can be executed more efficiently by merging the read and write phases and completely removing the validation phase, thus eschewing most of the traditional optimistic techniques and associated overhead. A transaction whose turn to commit has not yet come is a speculative transaction, and it follows the ordered commit protocol.

Figure 5.1 compares traditional transactions (left) to preordered transactions (right). Under the traditional transactions model there are two disjoint sets: the set of serialized (already committed) transactions, and the set of active transactions. Any transaction from the active set can be the next transaction to commit and move from the active to

```
begin
$v \leftarrow$ read($stock$)
if $v \geq qty$ then
    $v \leftarrow v - qty$
    write($stock$, $v$)
    commit
else
    abort
```

a Traditional interface.

```
begin
$\Box \leftarrow$ read($stock$)
if is-true($\{\Box \geq qty\}$) then
    $\triangle \leftarrow \{\Box - qty\}$
    write($stock$, $\triangle$)
    commit
else
    abort
```

b LSD interface.

Figure 5.2: Simplified transaction that buys qty amount of an item from an e-commerce application using the (a) traditional API, and (b) LSD API. With the traditional API a committing transaction is bound to a particular stock value, e.g. 42, so to ensure serializability the value can not change after the read operation until the transaction commits. With the LSD API a committing transaction never observes a concrete stock value during its execution, so to ensure serializability the value can change as long as it remains $\geq qty$ until the transaction commits.

the serialized set. (In the example it is transaction T_B.) In contrast, under the preordered transactions model the active set is a subset of the serialized set: the serialization order of all transactions is already established. Pot enforces this order using ordered commits, and further uses transaction modes to distinguish active transactions between fast and speculative transactions, depicted as a pentagon and squares, respectively.

To summarize, we propose to execute requests concurrently by executing them as preordered transactions. Preordered transactions are serialized in a common order across replicas during the state machine replication's agreement phase, and then executed using Pot to ensure the agreed upon order is respected.

For a more in depth presentation of Pot, please refer to chapter 3.

5.1.2 LSD

Since the research described in this dissertation proposes to extract concurrency using speculative execution, achieving good performance depends on whether speculation is successful or not. For workloads where transactions seldom conflict with each other, speculation is successful by definition. We seek to also improve the success of speculation for transactions that conflict with one another.

Consider the example in Figure 5.2a that depicts a simplified portion of the TPC-C new order transaction, [64] which implements the action of buying a certain quantity qty of items. If the item's stock $\big(v \leftarrow \text{read}(stock)\big)$ is enough to fulfill the order $\big(v \geq qty\big)$, the stock value decreases by qty $\big(\text{write}(stock, v - qty)\big)$.

This example illustrates how the read/write interface fails to convey the semantics of the transaction to the system, e.g., how the transaction's behavior depends on the values it reads, or how it computes the values it writes. Instead, from the point of view of the system, transactions are a sequence of opaque read/write operations.

To understand how this can be a limiting factor, consider the situation where the current stock value is 42 (*stock*), and the quantity to order is one (*qty*). When the transaction issues the read(*stock*) operation, it returns the value 42 to the transaction. Since the system does not know what the transaction will do with the returned value, it must be conservative to account for all possible situations, e.g. the transaction only executing some operations depending on the returned value, or using the returned value to compute the value of a subsequent write. As a consequence, the concurrency control protocol (Pot) records the read operation so that the system can check that the *stock* object's value is the same (42) when the transaction attempts to commit. If meanwhile another transaction modifies *stock*, then the transaction that observed the now-stale stock value fails to commit because the modification invalidates branching decisions or computed values by the transaction that observed the value 42.

As this example shows, a central part of enforcing transaction isolation is ensuring that the state that a transaction observes (i.e. the values returned by read operations) remains unchanged throughout its execution. Our key insight is to question whether a transaction *really* needs to observe a specific state during its execution. In other words, in our running example, does the read(*stock*) operation really need to expose a particular state to the transaction, e.g. 42? With the current interface the answer is yes. Otherwise, transactions cannot have conditional branches that depend on the system's state, nor perform updates to the state that are a function of itself. Going back to our example, the transaction could not check whether there is enough stock nor compute the new stock value.

In this research we get around these limitations by rethinking the transactional API in order to provide transactions that allow for greater concurrency. We call our rethought interface LSD, short for **l**azy **s**tate **d**etermination.

The key observation behind LSD is that, frequently, transactions do *not* need to observe a concrete state to execute most of their logic. Thus, we propose alternative semantics for the read operation. Specifically, the read operation should *not* expose a specific system state to the transaction by returning a concrete value, but will instead return a *future*. [7]

A future is an object that acts as a proxy for a value that is initially unknown. In our case, a future symbolizes the value of a specific state object. This means that the system *promises* the transaction to resolve the future's value, but does not do it right away. In particular, we want to defer evaluating futures until the transaction attempts to commit (*lazy evaluation* [36]) to maximize concurrency. (Note that the traditional semantics of the read operation is equivalent to returning futures that are immediately resolved.) Returning to our running example, we depict this modification in Figure 5.2b, and represent the future that symbolizes the stock value by □.

The proposed change to the semantics of the read operation has a clear benefit: if a transaction does not observe a specific system state, other transactions can modify it without breaking the isolation guarantees of the first transaction. However, this raises the problem of determining how can a transaction use futures. This can, in turn, be split into two main challenges. The first is how can a transaction perform conditional branching based on futures. The second is how can a transaction compute values that depend on futures. For instance, how can the logic of our example transaction decide whether it can fulfill the order if it does not know the stock value, and how can the transaction compute the new stock value? (A naive approach is to eagerly resolve futures when a transaction requires their value, but this again results in restricting concurrency.)

To solve the first challenge, we observe that a future symbolizes the value of a particular object. While we would like that a transaction is not able to directly observe the value of a future, we can still ask the system whether a future's value respects a certain condition. For example, the transaction can ask the system whether the stock value is greater than qty, and make a control flow decision depending on the system's answer.

To support this functionality we introduce a new operation, is-true(p), which, given a predicate function p over one (or more) futures, returns whether the predicate holds or not. We show the is-true operation using the $\{\square \geq qty\}$ predicate in Figure 5.2b. Note that while the is-true operation effectively exposes system state to the transaction, it exposes an abstract state (the stock is greater than qty) rather than a concrete one (the stock is 42), which has the potential to allow for more concurrency, e.g. by allowing concurrent modifications of the stock value as long as it remains $\geq qty$ after all the modifications.

The second challenge is how can a transaction perform computations using futures. To solve this challenge, we observe that while a transaction cannot perform the actual computation with futures, it can define the necessary computation and let the system perform it when the transaction commits and the futures resolve to concrete values. For example, the transaction can define that the new stock value is whatever value its future ends up resolving to minus qty.

To support this behavior we change the semantics of the write operation so that, instead of receiving the concrete new value for an object, it receives a function that computes the concrete value when evaluated. This function has the important property that it can depend on the values of any future, since the system knows how to resolve them. Furthermore, write functions are lazily evaluated by the system when the transaction commits, so that the futures that the functions depend on may remain unresolved. In Figure 5.2b, we represent this function as $\{\square - qty\}$. The argument of the write operation is the unevaluated function $\{\square - qty\}$. The system evaluates the function and applies its value when the transaction commits.

The proposed changes to the semantics of the read and write operations and the addition of the is-true operation allow the system to provide transactions whose speculation is more successful than was possible before, for two main reasons.

The first is that we decrease the duration of time that a transaction requires isolation. With the traditional interface, the transaction requires isolation from the moment when it first observes system state (with the traditional read operation semantics) until the transaction attempts to commit. In contrast, with LSD, the transaction only requires isolation during its commit operation if it does not require any specific conditions.

The second reason is that we relax the set of concurrent transactions that are forced to abort when executing concurrently with some transaction to guarantee serializability. Even when a transaction t needs to test some predicate over objects, LSD's is-true operation still allows concurrent transactions to modify those objects without requiring t to abort as long as these modifications do not invalidate t's previously asserted predicates. This contrasts with the traditional interface that prevents any modifications, whether they violate such predicates or not.

That said, it is important to stress that the LSD API is not a panacea. Transactions that must observe a concrete state can not reap LSD's benefits and need to resolve the required futures, falling back to the standard read semantics, e.g. because they externalize values to the user, or require concrete states such as $stock = 42$. However we believe that a large class of transactions are able to take advantage of LSD.

To summarize, we propose to alter the transactional API to expose more information about the transactions' semantics to the system, in order to increase the success of speculation. The new API allows transactions to execute against an abstract state, which allows for more concurrency without invalidating speculation. Transactions delay computations that depend on the abstract state until commit time, when the abstract state materializes into a concrete state over which the computations are performed.

For a more in depth description of LSD, please refer to chapter 4.

5.2 Combining Pot and LSD

A Pot speculative transaction only commits after being promoted to fast mode. The promotion can happen either when the transaction is waiting for its turn to commit, or at runtime. Regardless of when the promotion happens, a Pot transaction performs two steps when it transitions to fast mode. The first step is to validate whether the speculation that the transaction performed while in speculative mode still holds, i.e. none of the elements in the read set was modified since the transaction observed their values. The second and final step is to write back the values the transaction buffered in its write set while in speculative mode.

Figure 5.3 shows the transaction protocol that combines Pot and LSD. Essentially, with the addition of LSD, the Pot+LSD transaction performs three steps when it transitions to fast mode. Like the Pot-only transaction, the first step is to validate whether the speculation that the Pot+LSD transaction performed while in speculative mode still holds. In the case of a Pot+LSD transaction, besides validating that the read set remains valid, the Pot+LSD transaction *additionally* checks whether the transaction's condition set remains valid, i.e. evaluation its conditions return the same result that the transaction observed while in speculative mode. For example, if the transaction performed some actions during its speculative execution that depended on tx-is-true($\{\square \geq 0\}$) returning true, then the reevaluation of $\square \geq 0$ must return true again for the speculative execution to be correct.

```
tx-begin
    get sequence number
tx-read($key$)
    generate future □
    add □ to $R_f$
    return □
tx-read(△)
    resolve △ to $\langle value, version \rangle$
    add $\langle value, version \rangle$ to $R$
    return tx-read($value$)
tx-write($key$, □)
    add $\langle key$, □$\rangle$ to $W$
tx-write(△, □)
    add ⟨△, □⟩ to $W_f$
tx-is-true(□)
    $rvalues$ ← values necessary to resolve □
    resolve □ to $result$ using $rvalues$
    add ⟨□, $result\rangle$ to $C$
    return $result$
tx-commit
    wait for turn
    // fast mode's signal handler is executed
    // fast mode's tx-commit is executed
```

a Pot+LSD transaction in speculative mode.

```
when turn is signaled
    // step 1: validation
    validate $R$
    validate $C$
    // step 2: resolution
    resolve $R_f$
    resolve $W$
    resolve $W_f$
    // step 3: write back
    write back $W$
    write back $W_f$
tx-read($key$)
    generate future □
    return □
tx-read(△)
    resolve △ to $\langle value, - \rangle$
    return tx-read($value$)
tx-write($key$, □)
    resolve □ to $value$
    write $value$ to $key$
tx-write(△, □)
    resolve △ to $key$
    resolve □ to $value$
    write $value$ to $key$
tx-is-true(□)
    $rvalues$ ← values necessary to resolve □
    resolve □ to $result$ using $rvalues$
    return $result$
tx-commit
    signal next transaction's turn
```

b Pot+LSD transaction in fast mode.

Figure 5.3: The Pot+LSD transaction protocol. Figure 5.3a shows the speculative mode. Figure 5.3b shows the fast mode. R denotes the read set. W denotes the write set. R_f denotes the future read set. W_f denotes the future write set. C denotes the condition set. The protocol steps that pertain to Pot are highlighted.

The second step is the resolution step. In this step the Pot+LSD transaction resolves all the futures in the future read set, write set, and future write set. These futures must be resolved to concrete values before the transaction can apply its updates to the state. For example, if the transaction increments a counter, it performs □ ← tx-read($counter$) followed by tx-write($counter$, {□+1}). Resolving the future read set evaluates the counter's future into a concrete value, e.g. □ = 42. Resolving the write set evaluates the counter's new value function to 43, i.e. □ + 1 = 42 + 1 = 43.

The third and final step, like the Pot-only transaction, is to write back the values that the transaction buffered while in speculative mode. Those values were kept abstract (as futures and lazily evaluated functions) while the transaction was in speculative mode, but have been resolved by the previous step into concrete values that can be written back.

Besides the difference in the promotion algorithm, a Pot+LSD transaction in fast mode resolves the futures passed to its tx-write operations, so that it can immediately perform the corresponding writes.

5.3 Evaluation

We implemented a transactional key-value store prototype. The prototype is implemented as a Seastar [59] server, and data is stored in memory. Clients submit requests to the server over the network, which executes the requested commands as transactions. The transactions are implemented using the typical API (tx-begin, tx-read, tx-write, tx-commit, and tx-abort operations) or LSD's API which features our proposed tx-read, tx-write, and tx-is-true operations.

We implemented several concurrency control protocols as baselines. The Sequential baseline executes transactions one at a time. It represents the classical fault-tolerant state machine replication approach, where servers execute every transaction sequentially. Next, we implemented the classical OCC and 2PL baselines. These represent a typical fault-prone, unreplicated, server which executes transactions concurrently in a non-deterministic fashion. Finally, we implemented Pot and a combination of Pot with LSD (Pot+LSD). Pot represents our first contribution (chapter 3) that uses speculative concurrency to improve the execution phase performance of a fault-tolerant, state-machine-replicated, server. Pot+LSD represents our full vision which improves the effectiveness of Pot's speculation using the LSD API (chapter 4) to avoid conflicts.

5.3.1 Setup

We evaluated our prototype using the TPC-C benchmark with two workload configurations: one with lower, and one with higher contention.

The evaluation was run on a private gigabit ethernet cluster. The server runs on a NUMA machine with two 6-core Intel Xeon E5-2620 processor and 64GB of RAM. Clients run on the various remaining machines with AMD and Intel processors, and communicate with the server over the network.

Each data point reports the average of 5 runs.

5.3.2 Low contention

We show the result of the lower contention workload first. This workload was achieved by setting the number of warehouses to 32. The number of warehouses is the parameter that governs the amount of contention in the workload: the higher it is, the lower the contention.

Figure 5.4 shows the results we obtained. The x-axis shows the number of clients submitting requests to execute commands, i.e. TPC-C transactions, to the server. The

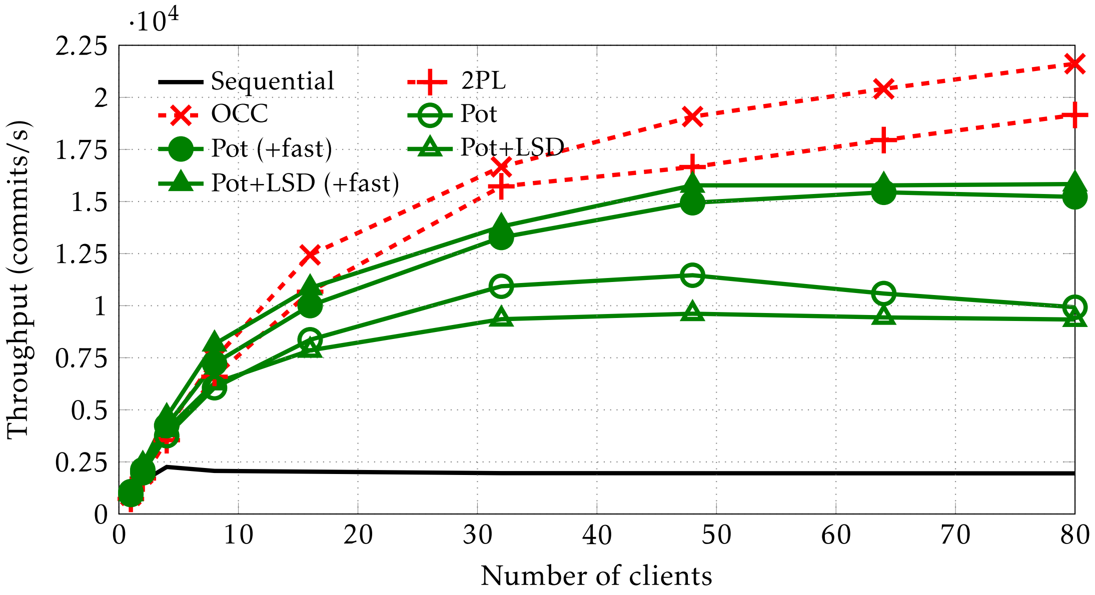


Figure 5.4: Performance of TPC-C using a workload with low contention (32 warehouses).

y-axis shows the server's throughput in tens of thousands of committed transactions per second.

Sequential. This series represents the classical fault-tolerant state machine replication approach, i.e. the server executes every transaction sequentially. This series is the baseline our contributions want to improve on.

As expected, its throughput remains flat when we increase the number of clients. The throughput we obtained with 80 clients was ≈ 2.5K committed transactions per second.

2PL and OCC. These series represent a typical fault-prone, unreplicated, server which executes transactions concurrently in a non-deterministic fashion using the 2PL and OCC protocols, respectively. These series serve as an upper bound to the gains we can expect from our contributions, because 2PL and OCC produce non-deterministic serialization orders.

As expected, by increasing the number of clients, the server's throughput also increases. The throughput we obtained with with 80 clients was ≈ 20K and ≈ 22.5K committed transactions per second when using 2PL and OCC, respectively. This is ≈ 8× the throughput we obtained with Sequential, which highlights the dire choice between performance and fault tolerance.

Pot. This series represents the first half of our contribution to improve the performance of a fault-tolerant, state-machine-replicated, server.

The throughput we obtained with 80 clients was ≈ 10K committed transactions per second. This result shows that the throughput of the fault-tolerant server using Pot is ≈ 2× worse than its fault-prone counterpart using 2PL/OCC. But considering that the Sequential baseline is ≈ 8× worse than 2PLL/OCC, this means that Pot was ≈ 4× better than Sequential. We consider this to be a worthwhile result, especially considering that it requires no additional burden from the application developer.

Pot+LSD. This series represents our full contribution to improve the performance of a fault-tolerant, state-machine-replicated, server. However, recall that the purpose of LSD is to improve the effectiveness of Pot's speculation in the presence of contention. This workload is a low contention one, so the results we obtained serve to show the price we are paying for LSD when it is not really needed, i.e. low contention scenarios.

The difference is most noticeable in the throughput we obtained with 48 clients, where Pot+LSD's throughput is $\approx 80\%$ of Pot's throughput. Although Pot+LSD imposed $\approx 20\%$ overhead over Pot, this workload is a worst-case scenario for LSD because there is little to no contention. In the next section we evaluate a higher contention workload, where Pot+LSD is $\approx 3\times$ better than Pot, which we argue make the $\approx 20\%$ overhead of Pot+LSD's in this workload an acceptable trade-off.

Pot (+fast) and Pot+LSD (+fast). These series explore what is the benefit of augmenting our approach with application-specific knowledge, by allowing the application developer to specify which transactions commute. Pot uses this information to execute consecutive commutative transactions in fast mode simultaneously.

For the particular case of our prototype, we specified that TPC-C transactions that operate over distinct warehouses are commutative—the warehouse identifier is an argument of every transaction. Taking advantage of this simple information about transaction commutativity led to Pot (+fast) and Pot+LSD (+fast) committing $\approx$ 15K transactions per second. This is $\approx 66\%$ of the throughput of the fault-prone 2PL/OCC series, and an improvement of $\approx 1.5\times$ over Pot and Pot+LSD.

5.3.3 High contention

In this section we show the result of the higher contention workload. This workload was achieved by setting the number of warehouses to 1.

Figure 5.5 shows the results we obtained. As in the previous Section, the x-axis shows the number of clients submitting requests to execute commands, i.e. TPC-C transactions, to the server. The y-axis shows the server's throughput in tens of thousands of committed transactions per second.

Sequential. The throughput of this series is indistinguishable from its throughput on Figure 5.4, i.e. $\approx$ 2.5K committed transactions per second. This is because Sequential executes transactions one a time, so the contention in the workload is irrelevant.

2PL. The throughput we obtained with with 80 clients was $\approx$ 10K committed transactions per second. This is $\approx 4\times$ the throughput we obtained with the classical fault-tolerant Sequential approach.

OCC. It is well known that OCC's optimistic approach is not suitable for high contention scenarios. The results we obtained corroborate this fact once again. The throughput we obtained with 80 clients was actually worse than Sequential, due to amount of wasted work OCC performs.

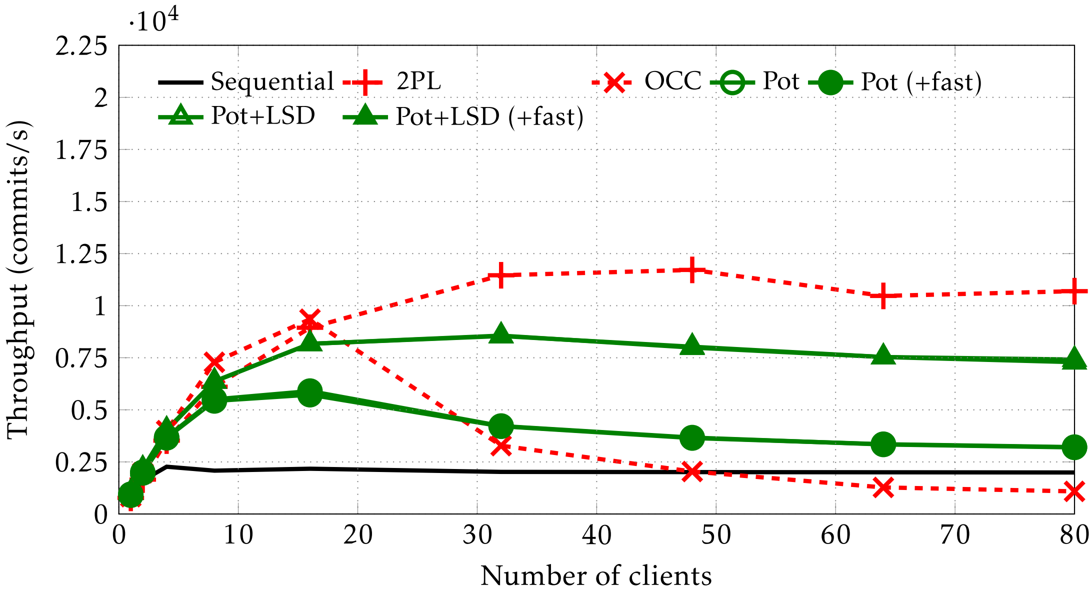


Figure 5.5: Performance of TPC-C using a workload with high contention (1 warehouse).

Pot. The throughput we obtained with 80 clients was ≈ 2.5K committed transactions per second, which is very close to Sequential. However, recall that Pot uses speculative techniques that struggle in the presence of contention, which is the case of this workload. The second half of our contributions, LSD, aims to improve Pot in these situations.

Pot+LSD. By combining Pot with LSD, the throughput we obtained with with 80 clients was ≈ 7.5K committed transactions per second. This is ≈ 3× better than Pot and Sequential, and is ≈ 75% of the throughput achieved by 2PL, which we argue is a great result.

Pot (+fast) and Pot+LSD (+fast). For this particular experiment, TPC-C is configured with a single warehouse. These series produced virtually the same results as Pot and Pot+LSD. This is expected, because all transactions operate over the same warehouse and are therefore deemed as non-commutative, which does not provide any improvement over Pot and Pot+LSD.

Chapter 6

Related work

This chapter contrasts the contributions of this dissertation against related work. Section 6.1 presents related techniques to achieve deterministic execution of multithreaded programs. Section 6.2 discusses related work that also enables concurrent request execution in the state machine replication model. Section 6.3 discusses other approaches to concurrency control and improvements to the transactional API to reduce conflicts. Each Section contrasts relevant related work with the contributions in this dissertation.

6.1 Deterministic multithreading

Many deterministic multithreading systems for programs using lock-based synchronization have been proposed, such as Kendo, [53] CoreDet, [9] Dthreads, [45] and Parrot. [15] At their core, they all follow a similar approach to guarantee determinism, which is to ensure that threads acquire locks in a total order that is deterministic across reexecutions. OptSCORE [33] is an example of a system that applies a deterministic multithreading system for lock-based synchronization to state machine replication.

Grace [10] ensures deterministic execution of programs that specifically use a structured fork-join style parallelism. Essentially, it executes the entire work of each thread as a transaction, and forces the transactions to commit in thread-creation order.

DeSTM [56] targets programs that use Software Transactional Memory (STM). It adapts the double barrier technique used by many deterministic lock-based systems to STM. In a nutshell, threads execute in rounds, where a round is comprised of one transaction per thread. A thread can only begin its transaction of round n when all the other threads have committed their transactions from round $n-1$, and a thread can only commit its transaction of round n when all the other transactions have started their transactions from round n—see Figure 3.9.

Discussion. If the transactional concurrency control protocol is implemented using locks, deterministic transactions could be implemented using the aforementioned deterministic multithreading system for lock-based programs. However this approach has several drawbacks. First, it cannot be applied to off-the-shelf Hardware Transactional Memory (HTM), because the concurrency control is implemented in the hardware. Second, it fails to exploit the semantics of transactions to reduce the overhead of ensuring determinism, because determinism is enforced by the lock acquisition order, and the locks are at a lower level of abstraction than transactions. Pot does not suffer from these drawbacks. It is applicable to both STM and HTM, and it operates at the transaction level, and employs its fast transaction mode to reduce overhead.

Compared to Grace, Pot is not restricted to the structured fork-join style parallelism, and takes advantage of the deterministic order to improve efficiency via fast transactions.

One of the key differences between DeSTM and Pot is that in Pot the sequencer establishes a deterministic transaction serialization order that is enforced, i.e., the final outcome is as if transactions executed in the serial order defined by the sequencer. DeSTM, on the other hand, uses a token-passing scheme that defines a deterministic order in which threads attempt to commit transactions. Thus, the final outcome is always equivalent to the same transaction serialization order, although that order is unknown beforehand. As a consequence of this design, DeSTM orders both aborts and commits and requires conflicts to be deterministic. Pot only orders commits, and works whether conflicts are deterministic or not. Pot's design allows it to achieve better performance than DeSTM, which is important for the objective of this dissertation's research: improving performance. We argue that requiring deterministic conflicts is a major drawback, since most STM implementations and all existing HTM processors suffer from false conflicts. To actually achieve deterministic execution, the DeSTM authors had to disable Address Space Layout Randomization, an operation system security technique involved in the protection of buffer overflow attacks.

6.2 State machine replication

Below we discuss related work that also allows replicas to execute requests concurrently. We group the related approaches in two broad categories: those that rely on application-specific information, and those that do not.

6.2.1 Using application-specific information

In the first category we have systems that require application-specific information about commutativity between operations to decide which operations shall execute concurrently.

CBASE [39] proposes to add a parallelizer phase between the agreement and execution phases. In the parallelizer phase, each replica individually analyses the incoming requests according to application-specific information to identify operations that can execute concurrently. If the information deems that two (or more) operations commute, then the system executes them concurrently.

Eve [37] also relies on a parallelizer to identify commutative operations. But unlike CBASE, Eve tolerates an incorrect parallelizer by running an additional verification phase after the execution phase. Replicas compute a digest of their state as part of the execution phase, an then run an agreement protocol to agree on a digest (and therefore final state) deemed correct. All replicas rollback and execute the operations sequentially if the verification phase is unable to reach agreement on a final state. Otherwise, the correct replicas transfer their state to the replicas that diverged.

Marandi et al. [48, 49] and Alchieri et. al [1] propose to partition the replica state. Operations are mapped to the partitions they access, and the system executes operations that access disjoint partitions concurrently.

Opt-PSMR [47] also relies on mapping operations to the partitions they access, but allows developers to define a conservative mapping that must be correct and a more aggressive, optimistic, mapping that may not always be correct. Replicas tolerate wrong optimistically-mapped operations by running an application-specific "safety check" procedure before execution. The procedure decides whether the optimistic mapping is correct given the operation in question and the current replica state. If not, the operation is resubmitted to the agreement phase and falls back to the conservative mapping.

Discussion. Distilling the systems presented in this section gives us their common core idea: an oracle is responsible for identifying commutative operations, and the system relies on the oracle's output to know which operations to execute concurrently. The concrete oracle all the systems propose is the developer. In general, it is unclear how realistic it is for developers to accurately identify commutative operations, or specify the safety checks. While an automated oracle is possible in theory, e.g., via static analysis techniques, [19] it remains unclear how general and feasible an automated approach is. The research in this dissertation proposes to execute operations concurrently using speculative techniques. This allows the system to execute operations concurrently without necessarily relying on the developer. Nonetheless, the proposal in this dissertation may still rely on the developer as an optimization—e.g. Pot (+fast) and Pot+LSD (+fast) series on Figures 5.4 and 5.5.

6.2.2 Without using application-specific information

In the second category we have systems that do not require application-specific information about the operations to execute them concurrently. The proposal in this dissertation plan falls in this category.

Calvin [63] identifies commutative operations using static analysis to identify their read and write sets before execution. Knowing these sets before execution allows Calvin to acquire all the appropriate locks for an operation before its execution. Commutative operations execute concurrently because they do not perform conflicting lock acquisitions, while operations that conflict on some object are serialized during the lock acquisions. For operations whose read and/or write sets are dynamic, Calvin requires the developer to specify, for each operation, a "helper" read-only query that performs the necessary reads to discover the full read and write sets. When Calvin receives an operation whose read and/or write set is dynamic, Calvin executes the "helper" query and submits the operation to the agreement phase. When the operation reaches the execution phase, Calvin checks whether the "helper" correctly identified the read and write sets. If so, Calvin proceeds to execute the operation; otherwise, the "helper" is re-executed and Calvin re-submits the operation to the agreement phase.

Rex [32] requires developers to implement thread-safe operations using locks. Rex follows an "execute-follow" approach where one replica, the primary, executes operations concurrently and collects a trace of the order in which the operations acquire locks. The trace is shipped to the other replicas, the secondaries, and each replica uses the trace to grant the locks in the same order when executing the operations, effectively following the primary.

Kim et al. [38] briefly point out that they allow developers to specify the order in which they want the operations to commit. However, no information is provided as to how this is done, nor how to do it efficiently.

Discussion. The approach of Calvin and Opt-PSMR [47] is the same, but Calvin is a specific realization of the approach in the context of databases. (Knowing the full read and write set allows one to define the conversative mapping in Opt-PSMR, Calvin's "helper" is equivalent to the optimistic mapping in Opt-PSMR, and Calvin's correctness test of the "helper" is akin to Opt-PSMR's "safety check.") In the context of databases, an automated oracle that is able to identify if two operations commute is feasible because objects are accessed in a structured way, e.g. accessed by primary key. But in a general-purpose system, objects may be accessed by following heap pointers which makes the feasibility of an automated oracle questionable. This dissertation's research proposes an approach that is amenable to both the context of database and general-purpose systems, where designing an automated oracle can range from difficult to impossible.

Rex is a sort of hybrid active-passive replication scheme where, despite the fact there is redundancy of computation because secondaries also execute operations, failures in the computation of the primary are not tolerated by the system. For instance, if the primary ships a trace that "does not make sense" to the secondaries, it appears that the secondaries would hang, or crash. (Rex's paper does not discuss these situations.) Unlike Rex, this dissertation's research proposes an approach where every replica is equivalent so there are no central points of failure.

Finally, we propose an approach akin to what Kim et al. describe. But the research in this dissertation develops the idea fully, not only detailing how to ensure that operations execute concurrently in such a way that is equivalent to the order that all replicas agreed upon, but also how to do so efficiently.

6.3 Transactions

Given that the idea in this dissertation's research is to execute operations concurrently as transactions under the state machine replication model, it is important to revisit their API and concurrency control protocols to ensure that we are able to reap the available concurrency as much as possible. Below we discuss related work about concurrency control protocols and related work that revisits the API to achieve more concurrency.

6.3.1 Concurrency control

Timestamp ordering [11] assigns each transaction a timestamp. When a conflicting access occurs, the concurrency control protocol aborts the transaction with the lowest timestamp to resolve the conflict. Whenever a transaction aborts it restarts with a more recent timestamp.

Discussion. At a first glance it may appear that timestamp ordering can be made to provide the same guarantees as Pot by assigning transactions the "right" timestamps that would remain unchanged despite aborts. For example, consider an agreed-upon order of $T_a \rightarrow T_b \rightarrow T_c$. It seems that assigning $timestamp(T_a) = 3$, $timestamp(T_b) = 2$, and $timestamp(T_c) = 1$, would enforce the desired serialization order. However that is not true, because the serialization order that timestamp ordering computes is, just like the other traditional concurrency control protocols, dependent on the nondeterministic interleaving that occurs at runtime between transactions. For instance, imagine that T_a reads, and T_b writes, object x. If T_a issues its read of x before T_b issues its write to x, then T_b will abort if it attempts to write x before T_a commits. In this case the protocol ensures $T_a \rightarrow T_b$. However, if T_b issues its write to x and commits before T_a attempts to read x, then T_a's read will observe T_b's write. This situation implies $T_b \rightarrow T_a$, which violates the agreed-upon order.

6.3.2 Increased concurrency

Salt [69] exposes more concurrency using nested transactions. It requires developers to decompose top-level ACID transactions into alkaline subtransactions, whose modifications become visible to other alkaline subtransactions after they commit, but before the top-level ACID transaction commits. Callas [70] automates Salt's methodology using complex static analysis and an iterative process to find good transaction decompositions using heuristics.

Quro [71] is a compiler that reorders the transaction logic to issue contentious operations as late as possible in the logic, i.e. right before the commit operation. Quro requires an initial profiling run to identify which operations are contentious using heuristics. The reordering has the potential to decrease the contention window between transactions.

A family of approaches such as versioned boxes, [13] transactional boosting, [34] CRDTs, [61] Bumper, [21] and Doppel, [51] extract concurrency by offering higher level update operations in the API (e.g. an increment operation). Transactions should use these operations instead of performing read-modify-write patterns. The idea is that transactions that conflict due to some read-modify-write operation on the same object can be made to commute by using the higher level operations instead.

Discussion. Salt requires developers themselves to decompose ACID transactions into alkaline subtransactions to extract concurrency. This requires a deep understanding of the difficult subtleties of concurrent programming in order to do safely. In contrast, this dissertation's research proposes to extract concurrency by enriching the API in a way that does not require any fundamental increase in complexity to the developer to use the API.

Callas requires complex static analysis of the logic of all transactions, and Callas' final transaction decomposition is ultimately tied to a particular workload. If the workload changes, or if new transactions are added, the entire process needs to be redone. This dissertation's proposal allows the system to extract concurrency at runtime, so it does not rely on any static analysis. Furthermore, it is dynamic: new transactions can execute safely at any time.

Quro also suffers from drawbacks of being a static approach, e.g. requires reprofiling and recompiling every time the set of transactions changes. Furthermore, Quro may not be able to reorder contentious operations due to data dependencies. LSD with its future and functions can behave as a "version" of Quro that is able to reorder contentious operations until commit time even when there are data dependencies, by delaying both the updates (LSD functions) and the data on which they depend (LSD futures).

The effectiveness of the approaches based on higher level operations depends on at least two factors.

The first factor is that there must exist suitable operations to allow transactions to replace their contentious read-modify-write patterns with higher level operations. For instance, to use such approaches successfully on the example of Figure 4.1a, we need an operation that can encapsulate the read of the stock value, the check of whether there is enough stock, and the update of the stock value $\big($e.g. decr_x_if_gte_y$(id, x, y)\big)$. Other scenarios will require their specific operations. Such an API can quickly become unwieldy. LSD provides futures and a set of operators to be used inside LSD functions instead. This allows transactions to compose futures and the available operators to specify their logic. Transactions can obtain the same result of the approaches based on higher level operations using LSD's building blocks without the need of an overly complex API.

The second factor is that the transactions' logic must not require observing contentious state. In this case, higher level operations can not help because the execution of the transaction logic will be tied to the particular state it observes. For example, consider the TPC-C benchmark's [64] new-order transaction. In it, new orders are assigned a monotonically increasing numeric identifier, which is used not only to identify the new order in question, but also in other subsequent writes. In order to use the identifier in the subsequent writes, the transaction *needs* to observe the concrete identifier to pass it along to the writes. The approaches based on higher level operations are unable to extract concurrency in this example because they require transactions to operate over a *concrete* state. LSD allows transactions to operate over an *abstract* state (a future of the identifier) and even pass it along to writes inside LSD functions.

Chapter 7

Conclusion

In this chapter we conclude the dissertation with some final remarks about the contributions described in the previous chapters, and outline future research directions that build atop the research reported in this dissertation.

In this research we investigate how to improve the performance of systems that tolerate faults, because fault tolerance is paramount due to the increasing scale at which computing systems operate. Concretely, we focus on fault-tolerant systems that use the state machine replication methodology. We propose to improve the performance of a state-machine-replicated system by taking advantage of multicore processors to execute multiple requests concurrently. However, there is an inherent tension between the state machine replication approach to fault tolerance and the ability to exploit the underlying hardware parallelism in todays multicore computer architectures. This tension exists because, on one hand, the state machine replication paradigm has a safety requirement that the outcome of the execution of requests has to be deterministic. On the other hand, concurrent execution is fundamentally non-deterministic. In this research we tackle this tension and propose a solution to it. In a nutshell, our solution is to execute requests concurrently, but control their concurrent execution in a way that ensures that the outcome is deterministic, as required by the state machine replication model.

The main challenge of our solution is *how* to control the execution to ensure a deterministic outcome. This challenge is addressed by our first contribution, Preordered transactions (Pot). The idea behind Pot is to execute the requests as speculative transactions that respect the state machine replication model, i.e. appear to execute sequentially in the agreed-upon order despite executing concurrently. To understand how Pot achieves this, it is important to remember that the concurrent execution of transactions is mediated by a concurrency control protocol. A typical concurrency control protocol ensures that, although transactions execute concurrently, the final outcome is as if they executed one at a time in some order—the keyword being *some*. Basically, any order is acceptable, so if the replicas execute the requests (transactions) concurrently, it is entirely possible that each replica may produce a different final outcome. Pot proposes a new concurrency control protocol that ensures that the final outcome of executing a set of transactions concurrently is as if the transactions executed one at a time *in a particular order*. In this case, that order will be the one that the replicas agreed on, hence respecting the state machine replication's safety requirement.

Succinctly, Pot achieves its goal by executing transactions speculatively, but committing their effects in the predefined order. The effectiveness of this approach is tied to the success of the speculative execution. If speculation fails constantly, we cannot expect to reap any performance benefits, so it is desirable for the speculation to succeed. Improving the success of the speculative execution is the objective of our second contribution, Lazy State Determination (LSD). LSD provides an enhanced transactional API that permits transactions to execute their logic over an abstract state, which increases the success of speculation. For example, consider a transaction that implements the withdrawal of money from a bank. Using the typical API, the transaction first observes what the current balance is, and if there is enough money, subtracts the withdrawal amount from the observed balance to calculate the new balance, and updates the balance to its new value. If some other transactions modifies the balance after the withdrawal transaction observes it, the computation performed by the withdrawal transaction will be incorrect, and thus its speculation unsuccessful. For instance, imagine that the transaction wants to withdraw 10€ and that it observes a balance of 100€. Meanwhile, the balance is increased to 110€. The withdrawal transaction will compute the new balance to be 90€: the balance it observed, 100€, minus the amount to withdraw, 10€. Updating the balance of 90€ effectively erases the 10€ increase that occurred meanwhile, which is clearly wrong. Hence, the speculation was unsuccessful.

To understand how LSD can help improving the success of the speculative execution, note that in our withdrawal example the transaction needs to observe the current balance do decide if there is enough money. Unfortunately, as soon as the balance is observed, any modification to its value will cause the transaction's speculation to fail. With LSD, instead of observing what is the actual balance, the transaction can *ask* the system if there is enough balance to withdraw, i.e. if the current balance is greater than 10€. At this point, with LSD, the transaction's speculation only depends on the balance being greater than 10€, which allows more concurrency than depending on the balance being *exactly* 100€.

The next important step is the part where the withdrawal transaction computes the new balance value. If the transaction needs to observe the balance to compute the new balance value, we are back to square one. Thus, with LSD the transaction can tell the system *how* to compute the new balance value, i.e. *whatever the balance is* minus 10€, and the system will perform that computation *when the transaction commits*. This running example shows how LSD can improve speculation by executing its logic over an abstract state—the balance is greater than 10€, the new balance will be whatever it currently is minus 10€—instead of a concrete state—the balance is 100€, the new balance will be 90€.

An experimental evaluation of a prototype implementation of Pot and LSD (Section 5.3) shows that the contributions in this dissertation can improve the performance of a fault-tolerant server up to ≈ 4×. From other point of view, the experimental results show that a fault-tolerant server can go from being ≈ 8× slower (using sequential execution) to only ≈ 25% slower (using concurrent execution with Pot and LSD) than a concurrent fault-prone server. We argue that these results demonstrate the validity of our thesis: it is possible to improve the performance of the state machine replication's execution phase by taking advantage of multiple processors to execute operations concurrently, while at the same time respecting the safety properties of the state machine replication approach.

Future research directions

Pot can execute multiple transactions in fast mode if the transactions commute with each other. One avenue of future research is to devise automatic methods to verify if two transactions commute. This will allow a system using Pot to improve its performance further, without requiring additional effort from application developers.

Other possible research direction is to formally specify the semantics of the LSD API and its adapted concurrency controls. For one, this will permit a formal proof that LSD provides serializability, by establishing that the LSD-aware concurrency control protocols are equivalent to their original counterparts.

Formally specifying LSD can also aid in devising automatic translation methods that are able to take in an existing application, and produce a modified version of the application that uses the LSD API where appropriate.

Another direction is to investigate how to integrate LSD into a relational database and the ubiquitous Structured Query Language (SQL), which is the most widely used type of database, and interface between applications and the database, respectively.

Bibliography

[1] E. Alchieri, F. Dotti, and F. Pedone. “Early scheduling in parallel state machine replication.” In: *ACM Symposium on Cloud Computing (SoCC)*. 2018.

[2] P. A. Alsberg and J. D. Day. “A Principle for Resilient Sharing of Distributed Resources.” In: *International Conference on Software Engineering (ICSE)*. 1976.

[3] Amazon Web Services, Inc. *Amazon Elastic Compute Cloud*. https://aws.amazon.com/ec2. Mar. 2017.

[4] Amazon.com, Inc. *Amazon*. https://www.amazon.com. Mar. 2017.

[5] Apache Software Foundation. *Apache Thrift*. https://thrift.apache.org. Nov. 2016.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. “A view of cloud computing.” In: *Commun. ACM* 53.4 (2010). doi: 10.1145/1721654.1721672.

[7] H. Baker Jr. and C. Hewitt. “The incremental garbage collection of processes.” In: *SIGPLAN Not.* 12.8 (1977). doi: 10.1145/872734.806932.

[8] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. “Putting Consistency Back into Eventual Consistency.” In: *ACM European Conference on Computer Systems (EuroSys)*. 2015. doi: 10.1145/2741948.2741972.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. “CoreDet: A compiler and runtime system for deterministic multithreaded execution.” In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2010. doi: 10.1145/1736020.1736029.

[10] E. D. Berger, T. Yang, T. Liu, and G. Novark. “Grace: Safe Multithreaded Programming for C/C++.” In: *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2009. doi: 10.1145/1640089.1640096.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987. isbn: 0-201-10715-5.

[12] C++ Committee SG5. *Technical Specification for C++ Extensions for Transactional Memory*. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4514.pdf. 2015.

[13] J. Cachopo and A. Rito-Silva. "Versioned boxes as the basis for memory transactions." In: *Sci. Comput. Program.* 63.2 (2006). DOI: 10.1016/j.scico.2006.05.009.

[14] H. Cain, M. Michael, B. Frey, C. May, D. Williams, and H. Le. "Robust architectural support for transactional memory in the POWER architecture." In: *International Symposium on Computer Architecture (ISCA)*. 2013. DOI: 10.1145/2485922.2485942.

[15] H. Cui, J. Simsa, Y. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. Gibson, and R. Bryant. "Parrot: A practical runtime for deterministic, stable, and reliable threads." In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2013. DOI: 10.1145/2517349.2522735.

[16] L. Dalessandro, M. F. Spear, and M. L. Scott. "NOrec: Streamlining STM by Abolishing Ownership Records." In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2010. DOI: 10.1145/1693453.1693464.

[17] J. Dean. *Building software systems at Google and lessons learned*. https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf. Nov. 2010.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store." In: *SOSP*. 2007. DOI: 10.1145/1294261.1294281.

[19] R. J. Dias, D. Distefano, J. C. Seco, and J. M. Lourenço. "Verification of snapshot isolation in transactional memory Java programs." In: *European Conference on Object-Oriented Programming (ECOOP)*. 2012. DOI: 10.1007/978-3-642-31057-7_28.

[20] D. Dice, O. Shalev, and N. Shavit. "Transactional locking II." In: *International Symposium on Distributed Computing (DISC)*. 2006. DOI: 10.1007/11864219_14.

[21] N. Diegues and P. Romano. "Bumper: Sheltering transactions from conflicts." In: *IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 2013. DOI: 10.1109/SRDS.2013.27.

[22] A. Dragojević, D. Narayanan, E. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. "No compromises: distributed transactions with consistency, availability, and performance." In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2015.

[23] eBay, Inc. *eBay*. https://www.ebay.com. Mar. 2017.

[24] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. "The notions of consistency and predicate locks in a database system." In: *Commun. ACM* 19.11 (1976). DOI: 10.1145/360363.360369.

[25] Facebook. *RocksDB*. https://rocksdb.org. Nov. 2016.

[26] Facebook. *Facebook*. https://www.facebook.com. Mar. 2017.

[27] Free Software Foundation. *Transactional memory in GCC*. https://gcc.gnu.org/wiki/TransactionalMemory. 2014.

[28] S. Gilbertson. *Lessons from a cloud failure: It's not Amazon, it's you*. https://www.wired.com/2011/04/lessons-amazon-cloud-failure. Apr. 2011.

[29] Google. *Google Compute Engine*. https://cloud.google.com/compute. Mar. 2017.

[30] R. Guerraoui and M. Kapalka. "On the Correctness of Transactional Memory." In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2008. DOI: 10.1145/1345206.1345233.

[31] R. Guerraoui, M. Kapalka, and J. Vitek. "STMBench7: A Benchmark for Software Transactional Memory." In: *ACM European Conference on Computer Systems (EuroSys)*. 2007.

[32] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. "Rex: Replication at the speed of multi-core." In: *ACM European Conference on Computer Systems (EuroSys)*. 2014. DOI: 10.1145/2592798.2592800.

[33] G. Habiger, F. J. Hauck, J. Kostler, and H. P. Reiser. "Resource-efficient state-machine replication with multithreading and vertical scaling." In: *European Dependable Computing Conference (EDCC)*. 2018.

[34] M. Herlihy and E. Koskinen. "Transactional boosting: A methodology for highly-concurrent transactional objects." In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2008. DOI: 10.1145/1345206.1345237.

[35] M. Herlihy and J. E. B. Moss. "Transactional memory: Architectural support for lock-free data structures." In: *International Symposium on Computer Architecture (ISCA)*. 1993. DOI: 10.1145/165123.165164.

[36] P. Hudak. "Conception, evolution, and application of functional programming languages." In: *ACM Comput. Surv.* 21.3 (1989). DOI: 10.1145/72551.72554.

[37] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. "All about Eve: Execute-verify replication for multi-core servers." In: *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2012. ISBN: 978-1-931971-96-6.

[38] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. "Improving server applications with system transactions." In: *ACM European Conference on Computer Systems (EuroSys)*. 2012. DOI: 10.1145/2168836.2168839.

[39] R. Kotla and M. Dahlin. "High throughput byzantine fault tolerance." In: *International Conference on Dependable Systems and Networks (DSN)*. 2004. DOI: 10.1109/DSN.2004.1311928.

[40] H. T. Kung and J. T. Robinson. “On Optimistic Methods for Concurrency Control.” In: *ACM Trans. Database Syst.* 6.2 (1981). DOI: 10.1145/319566.319567.

[41] L. Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Commun. ACM* 21.7 (1978). DOI: 10.1145/359545.359563.

[42] L. Lamport. “The part-time parliament.” In: *ACM Trans. Comput. Syst.* 16.2 (1998). DOI: 10.1145/279227.279229.

[43] L. Lamport, R. Shostak, and M. Pease. “The byzantine generals problem.” In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982). DOI: 10.1145/357172.357176.

[44] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. “Warranties for faster strong consistency.” In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2014. ISBN: 978-1-931971-09-6.

[45] T. Liu, C. Curtsinger, and E. Berger. “DThreads: Efficient deterministic multithreading.” In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2011. DOI: 10.1145/2043556.2043587.

[46] S. Lu, S. Park, E. Seo, and Y. Zhou. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics.” In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008. DOI: 10.1145/1346281.1346323.

[47] P. J. Marandi and F. Pedone. “Optimistic parallel state-machine replication.” In: *IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 2014. DOI: 10.1109/SRDS.2014.25.

[48] P. J. Marandi, M. Primi, and F. Pedone. “High performance state-machine replication.” In: *International Conference on Dependable Systems and Networks (DSN)*. 2011. DOI: 10.1109/DSN.2011.5958258.

[49] P. J. Marandi, C. E. Bezerra, and F. Pedone. “Rethinking state-machine replication for parallelism.” In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2014. DOI: 10.1109/ICDCS.2014.45.

[50] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. “STAMP: Stanford transactional applications for multi-processing.” In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2008.

[51] N. Narula, C. Cutler, E. Kohler, and R. Morris. “Phase reconciliation for contended in-memory transactions.” In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014. ISBN: 978-1-931971-16-4.

[52] Netflix, Inc. *Netflix*. https://www.netflix.com. Mar. 2017.

[53] M. Olszewski, J. Ansel, and S. Amarasinghe. “Kendo: Efficient deterministic multithreading in software.” In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2009. DOI: 10.1145/1508244.1508256.

[54] P. E. O'Neil. "The Escrow Transactional Method." In: *TODS* 11.4 (1986). DOI: 10.1145/7239.7265.

[55] C. Papadimitriou. "The serializability of concurrent database updates." In: *JACM* 26.4 (1979). DOI: 10.1145/322154.322158.

[56] K. Ravichandran, A. Gavrilovska, and S. Pande. "DeSTM: Harnessing Determinism in STMs for Application Development." In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2014. DOI: 10.1145/2628071.2628094.

[57] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. "The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis." In: *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2015. DOI: 10.1145/2723372.2723720.

[58] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial." In: *ACM Comput. Surv.* 22.4 (1990). DOI: 10.1145/98163.98167.

[59] ScyllaDB. *Seastar*. http://seastar.io. Sept. 2018.

[60] S. Shankland. *Google spotlights data center inner workings*. https://www.cnet.com/news/google-spotlights-data-center-inner-workings. May 2008.

[61] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types." In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. 2011. DOI: 10.1007/978-3-642-24550-3_29.

[62] N. Shavit and D. Touitou. "Software transactional memory." In: *Distributed Computing* 10.2 (1997). DOI: 10.1007/s004460050028.

[63] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. "Calvin: Fast distributed transactions for partitioned database systems." In: *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2012. DOI: 10.1145/2213836.2213838.

[64] TPC. *TPC-C*. http://tpc.org/tpcc. Apr. 2017.

[65] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. "Speedy Transactions in Multicore In-memory Databases." In: *SOSP*. 2013. DOI: 10.1145/2517349.2522713.

[66] Twitter. *Twitter*. https://www.twitter.com. Mar. 2017.

[67] T. M. Vale, J. A. Silva, R. J. Dias, and J. M. Lourenço. "Execução concorrente e determinista de transacções." In: *Simpósio de Informática (INForum)*. 2015.

[68] T. M. Vale, J. A. Silva, R. J. Dias, and J. M. Lourenço. "Pot: Deterministic transactional execution." In: *ACM Trans. Archit. Code Optim.* 13.4 (2016). DOI: 10.1145/3017993.

[69] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. "Salt: Combining ACID and BASE in a distributed database." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014. ISBN: 978-1-931971-16-4.

[70] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. "High-performance ACID via modular concurrency control." In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2015. DOI: 10.1145/2815400.2815430.

[71] C. Yan and A. Cheung. "Leveraging lock contention to improve OLTP application performance." In: *PVLDB* 9.5 (2016). DOI: 10.14778/2876473.2876479.

[72] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. "Performance evaluation of Intel transactional synchronization extensions for high-performance computing." In: *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. 2013. DOI: 10.1145/2503210.2503232.