Improving Performance of Highly-Programmable Concurrent Applications by Leveraging Parallel Nesting and Weaker Isolation Levels

Duane F. Niles, Jr.

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Dongyoon Lee
Robert P. Broadwater
Roberto Palmieri

June 22, 2015
Blacksburg, Virginia

Keywords: Concurrency, Transactional Memory, Parallel Nesting, Distributed Systems, Databases, Transactions, Serializability, Weaker Isolation Levels

Improving Performance of Highly-Programmable Concurrent Applications by Leveraging Parallel Nesting and Weaker Isolation Levels

Duane F. Niles, Jr.

(ABSTRACT)

The recent development of multi-core computer architectures has largely affected the creation of everyday applications, requiring the adoption of concurrent programming to significantly utilize the divided processing power of computers. Applications must be split into sections able to execute in parallel, without any of these sections conflicting with one another, thereby necessitating some form of synchronization to be declared. The most commonly used methodology is lock-based synchronization; although, to improve performance the most, developers must typically form complex, low-level implementations for large applications, which can easily create potential errors or hindrances.

An abstraction from database systems, known as *transactions*, is a rising concurrency control design aimed to circumvent the challenges with programmability, composability, and scalability in lock-based synchronization. Transactions execute their operations speculatively and are capable of being restarted (or rolled back) when there exist conflicts between concurrent actions. As such issues can occur later in the lifespans of transactions, entire rollbacks are not that effective for performance. One particular method, known as *nesting*, was created to counter that drawback. Nesting is the act of enclosing transactions within other transactions, essentially dividing the work into pieces called *sub-transactions*. These sub-transactions can roll back without affecting the entire main transaction, although general nesting models only allow one sub-transaction to perform work at a time.

The first main contribution in this thesis is SPCN, an algorithm that parallelizes nested transactions while automatically processing any potential conflicts that may arise, eliminating the burden of additional processing from the application developers. Two versions of SPCN exist: Strict, which enforces the sub-transactions' work to be made visible in a serialized order; and Relaxed, which allows sub-transactions to distribute their information immediately as they finish (therefore invalidation may occur after-the-fact and must be handled). Despite the additional logic required by SPCN, it outperforms traditional closed nesting by 1.78× at the lowest and 3.78× at the highest in the experiments run.

Another method to alter transactional execution and boost performance is to relax the rules of visibility for parallel operations (known as their *isolation*). Depending on the application, correctness is not broken even if some transactions see external work that may later be undone due to a rollback, or if an object is written while another transaction is using an older instance of its data. With lock-based synchronization, developers would have to explicitly design their application with varying amounts of locks, and different lock organizations or hierarchies, to change the strictness of the execution. With transactional systems, the processing performed by the system itself can be set to utilize different rulings, which can change the performance of an application without requiring it to be largely redesigned.

This notion leads to the second contribution in this thesis: AsR, or *As-Serializable* transactions. *Serializability* is the general form of isolation or strictness for transactions in many applications. In terms of execution, its definition is equivalent to only one transaction running at a time in a given system. Many transactional systems use their own internal form of locking to create Serializable executions, but it is typically too strict for many applications. AsR transactions allow the internal processing to be relaxed while additional meta-data is used external to the system, without requiring any interaction from the developer or any changes to the given application. AsR transactions offer multiple orders of magnitude more in throughput in highly-contentious scenarios, due to their capability to outlast traditional levels of isolation.

This work is supported in part by US National Science Foundation under grant CNS 1217385, and NAVSEA/NEEC under grant 3003279297.

Acknowledgments

I would like to thank the following people, without whom my work would not be possible:

Dr. Binoy Ravindran, my advisor, for providing me with the opportunity to perform this research, and for his faith in my work.

Dr. Dongyoon Lee and Dr. Robert P. Broadwater, for graciously serving on my committee.

Dr. Roberto Palmieri, for his great knowledge and guidance in my research, which kept me optimistic and focused towards my goals.

All of the members in the Systems Software Research Group, for their friendship and assistance, including Sachin Hirve, Sean Moore, Alexandru Turcu, and Sebastiano Peluso.

Lastly, all of my friends and family, especially my parents and grandparents, for their support and love that allowed me to pursue this education.

Contents

List of Figures

Chapter 1

Introduction

Throughout the past decade, multi-core computer architectures have arisen as the main focus of hardware design and development. Until about 2004, traditional hardware expansion involved the ever-increasing density of transistors within processors and order-of-magnitude boosts in clock speed. For several decades, many developers worked towards a predicted rate of advancement, informally coined as *Moore's law* [1], which stated that the number of components compacted into integrated circuits would double every two years. While this prediction did roughly hold over time, computers were not to improve in the same way forever. Eventually, the development of singular processors reached a saturation point, brought on by various physical limitations, especially in terms of power consumption and heat. To continue moving forward with hardware design, system architects turned to a different paradigm of combining processors together into one chip, allowing each of them to work simultaneously, executing different threads of software.

Such a change in architecture greatly impacted the software side of the technology. In order to fully take advantage of multi-core systems, everyday application developers have had to adopt concurrent programming, designing applications to be split into sections able to execute in parallel, without any of these sections conflicting with one another. The complexity of parallelizing an application grows exceptionally with its size, making it an extensive obstacle in the path of improving system performance. Lock-based synchronization is generally the dominant form of this style of programming, which requires developers to explicitly mark objects and functions that may only be used by a limited number of threads at any instant of time—these locked sections are mutually exclusive between threads. While locking can be employed simplistically, by denoting large sections of code as needing to be synchronized, it can then become very inefficient, as the processors will not all be utilized.

Very detailed, low-level locking (i.e., covering the smallest amounts of items and functions possible, while still ensuring correct operation) is the best way to reap the benefits of parallel processing, and yet developing such code is largely problematic. Locking holds many drawbacks that can in fact make a program's parallel execution worse than its sequential form, and it can even render a program unable to continue (i.e., deadlock), if the locking is not programmed correctly. The programmability of locking has not entirely improved over the years and can still easily slow the design of an application. Often, because the interleaving of various threads in a program can execute in many different combinations, errors may not be noticed until long after a large application has been deemed complete. Due to the complexity, some developers may try to utilize higher-level locking to parallelize their code and ensure correctness more simply (but that can also make threads execute more sequentially). Even further, parallel applications are generally limited by what is known as *Amdahl's law* [2]. Amdahl predicts that any parallel program's speedup will hit an eventual, hard ceiling, based on the percentage of the program that is sequential, and regardless of the number of cores on its machine. Simply put, the larger the sequential portion of a program, the far less the program's speed can be enhanced.

Because of the low programmability of traditional concurrent programming, the concept of *transactions* was borrowed from the database community and has been reformed for parallel application development. A transaction can be understood simply as an indivisible set of operations of any magnitude. Relative to lock-based synchronization, a transaction is a mutually exclusive set of operations that do not directly prevent multiple threads from accessing them—but the actions are executed speculatively, i.e., they are not completed or externally visible until a set of rules are guaranteed to be met. Transactional rules vary entirely upon the system or methods being used, as well as the application being made; but if the set of operations follows those rules, then they are explicitly finalized. If any transactional rule is broken by any activity, then the conflicting transaction must undo everything that has been performed.

From this abstraction, the implementation of parallel code has become more straightforward, but the performance of such applications has not as easily improved. Although transactions have more programmability than low-level synchronization, the abstraction is so generalized and high-level that it cannot exploit a programmer's knowledge of an application that freely. Therefore, a trade-off between the programmability and efficiency of code arises as a crucial issue. The lower an application's primitives go into the code, the more efficient the application can be made; but it becomes more and more difficult to actually develop, eventually allowing the complexity to greatly outweigh the possible benefits. With higher-level abstractions like transactions, developers can program in a simpler manner; although the benefits of detailed program knowledge are left out if the primitives are not used at deeper levels.

In this sense, the basic form of transactions are too conservative for concurrent programming, and the design of a parallel program is still non-trivial. Hence, there exists great research to isolate the internal details of transactions from everyday programmers, while attempting to improve their parallelism as a whole. Following these goals, this research explores two pieces of orthogonal work: SPCN, an algorithm that enhances nested transactional applications by parallelizing their internal, lower-level transactions; and AsR, an algorithm which can be applied to any transactional system to enforce Serializable transaction rules through external processing, alleviating some of the system's internal overhead.

1.1 Summary of Thesis Research Contributions

The major contributions of this thesis are as follows:

- SPCN, an algorithm that can increase the parallelism of new or existing (composable) transactional applications without requiring any additional programmer action. Transactions can be composed inside of one another, which is known as using subtransactions; but only one sub-transaction is typically allowed to run at any given instant. While it is possible to parallelize these internal transactions, doing so allows for potential conflicts between them and can possibly stop programs from working properly, further complicating transactional programming. The given algorithm manages this parallelization automatically and also solves any potential conflicts that may occur, thereby keeping the programmer out of the loop while also improving the transactional performance.
- AsR, an algorithm that relaxes the typical constraint processing performed on transactions and attempts to consolidate the same transactional application rules through the usage of meta-data. For transactions to meet their highest expectations, i.e., to be serializable, they generally use more pessimistic processing than may be needed for some applications. If the commanding properties are altered, e.g., by tracking relevant action rather than explicitly locking data at every minor step, it is possible that more transactions can correctly finish over time than before, boosting the throughput of an application's work. With AsR, a layer is added over a given transactional system in order to perform higher-level processing that enforces the necessary rules while allowing the system to relax its internal processing.

While these two developments are in different aspects of transactional programming, they are not entirely unrelated, nor are they too fine-grained. As can be seen later, both designs can be implemented in systems with boundaries and setups different from those in which the current experiments were implemented. Further, the algorithms are easily adaptable and can be combined to increase the performance of different styles of applications.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 covers necessary background material relevant to the SPCN and AsR algorithms. In particular, it briefly summarizes the basic properties of transactions and their origin from databases, typical concurrent programming methods and drawbacks, as well as Transactional Memory (TM). Chapter 3 presents the SPCN algorithm, designed to parallelize nested transactions in new or existing transactional applications. In addition to the algorithm, background details for distributed systems as well as analysis of SPCN's efficiency are given. Chapter 4 presents the AsR algorithm, designed to provide Serializable transactions through processing that is external to the system used, allowing internal synchronization to be lightened. Further, the transaction property of isolation (as well as its rules and anomalies) is explored, and AsR's performance is evaluated. Chapter 5 reviews the conclusions formed from this work and some further developments that can extend the given research.

Chapter 2

Background

2.1 Databases and Transactions

Database Management Systems (DBMS) expanded greatly over many decades to become one of the most-used methods of concurrency in applications. In order for databases to perform operations on their data, *transactions* are utilized. From the system's internal perspective, a transaction can range from a small, singular operation to a large, complex set of operations. However, from an external perspective, transactions must operate as if they are singular actions themselves—they are indivisible. A typical example is a bank account transfer. Transfers utilize two different accounts, withdrawing a set amount of funds from the first and depositing the same amount into the second. If either the withdraw or deposit fail, then the transaction itself cannot complete.

Database transactions have four distinct properties altogether, coined as ACID [3]. The first property, *atomicity*, was briefly described above. An atomic transaction must be "all or nothing" (i.e., if any of its operations fail to complete correctly, then the transaction cannot complete). More explicitly, a transaction *commits* when all of its operations have successfully completed; and when a transaction commits, all of its operations' outcomes must be visible, or else the transaction is not atomic. If a transaction cannot complete all of its operations, it *aborts*, eliminating the fact that any work was attempted in the first place.

The next property is *consistency*, although it is the most abstract of the four properties. The basic definition requires that all transactions move the database system from one valid state to another valid state. Ambiguity lies in what makes a database's state *valid*, as that varies from system to system and application to application. Thus, constraints must be defined for a system, and if the constraints are guaranteed to be met, then the system's transactions are consistent. One simple constraint required by most (but not all) systems or applications is that transactions starting in the future, relative to previously-committed transactions, should always see the effects of those older transactions.

Isolation is related to the validity of a system and defines the time at which transactions' effects are visible. The most basic expectation for transactions is that their operations' outcomes cannot be seen until the transaction has successfully committed. However, it is actually possible for that to be relaxed. For instance, some applications may not require that rule, thus some of a transaction's work may be seen even though the transaction is still running (and can potentially abort). Relaxing this constraint generally allows for more concurrency in an application, as stricter isolation requires additional logic to be implemented (thus adding an overhead to the system). Isolation is expanded upon in Section 4.5 as it is a central piece of AsR (Chapter 4).

Lastly, *durability* rounds out the ACID properties. Arguably the simplest property, durability mandates that all committed transactions will always remain committed, even if the database faces issues such as crashes, logical errors, or power loss. The general approach to ensure a database's durability is to properly log all transactional work performed on the database and to flush the logs to non-volatile storage before transactions can commit.

2.2 Multi-Threaded Synchronization: Problems and Properties

Lock-based synchronization is a traditional methodology utilized to implement multi-threaded applications, in order to ensure operational correctness. Locking is applied to portions of an application (called *critical sections*) that cannot be run simultaneously by multiple threads. The previous example of a bank account transfer can be applied here as well. Reading and updating the balances of two bank accounts must occur in a properly synchronized moment of time. If not, it is possible for the balance of either account to change between the time a thread read and then updated it, meaning that information is no longer valid (i.e., funds could have been lost or even incorrectly added).

```
          Coarse             |         Fine
lock(accounts);              | lock(acc1);
acc1 = accounts.get(key1);   | lock(acc2);
acc2 = accounts.get(key2);   | acc1.withdraw(amount);
acc1.withdraw(amount);       | acc2.deposit(amount);
acc2.deposit(amount);        | unlock(acc1);
unlock(accounts);            | unlock(acc2);
```

Figure 2.1: Coarse-Grained vs. Fine-Grained Locking

Despite the importance of synchronization in applications, locking has a rather notable flaw. The efficiency and usability of locking can vary drastically depending upon the style of locking used and the application being developed. The simplest style is *coarse-grained* locking: data structures that can be used by multiple threads are locked until all relevant operations are completed. As shown in the left half of Figure 2.1, the structure used to access bank accounts is locked whenever a transfer is made. While this method of locking is very simple, it is poor in performance, as it explicitly serializes threads. If the entirety of a large data structure (operated upon by many threads) is locked, then concurrency serves little purpose, as only one thread can work while all other threads are unable to move forward.

On the other hand, data structures can be broken apart into smaller pieces, each of which are then synchronized individually. As seen in the right half of Figure 2.1, the bank accounts themselves can be locked (as opposed to locking the structure that holds all accounts). This method is known as *fine-grained* locking. While fine-grained locking enables more threads to perform concurrent work than coarse-grained locking can, implementing the synchronization is now more difficult. Lower-level processing is required, thus increasing the complexity of an application, which in turn affects the *liveness* [4, 5] of the application (i.e., its capability to make progress in reasonable amounts of time).

Various concurrency issues can easily occur and limit an application's liveness due to fine-grained locking. The most basic liveness problem is that of *deadlock*, where two or more threads overlap on some set of synchronized data such that none of the threads can make any progress. The right example shown in Figure 2.1 (for fine-grained locking) could allow deadlocking between two threads if they both require the same two bank accounts, but they access the accounts in opposite order. If one thread manages to lock `acc1` while the other locks `acc2`, then neither thread can continue, as they will wait indefinitely upon each other.

A simple solution to that problem would be for all threads to lock accounts in a specific order, i.e., to always lock the account with the lower account number first. Hence, both threads would go to lock `acc1` first, and only one of them would succeed. However, solving such deadlock problems is non-trivial in most applications, especially because not all problems may occur during testing. Further, fine-grained locking is generally not *composable*; it is difficult to collect various lock-based functions and combine them into a larger application function without first modifying them. In fact, it is nearly impossible to compose lock-based functions without at least knowing what is locked inside of them. Otherwise, it is rather easy for composed locks to create problems like deadlocking or breaking correctness.

Other liveness problems are *starvation* and *livelock*. Starvation, as the name implies, is when any number of threads cannot make progress simply because they are denied their synchronized resources over and over. More explicitly, a thread is "starved" if it continuously waits because the objects it needs are consistently allocated to other threads. It is actually easy for multiple threads to be starved in an application if there exists heavily-shared data that are used by a "greedy" thread. Such a thread is simply one that holds the data during very long operations, and it performs those same actions often.

Livelock is similar to deadlock but comes from threads constantly reacting to each other's actions, thereby not actually performing their own work. If threads have plans of action laid out but need to adjust depending upon what other threads have done (and will do), it is possible that they may spend more time adapting their plans than doing anything else. Further, livelock can occur in systems that attempt to detect deadlock, but only if multiple threads are restarted when a possible detection occurs.

Because of locking's many problems and complexity, recent decades have introduced the concurrency control of Transactional Memory (TM) to provide synchronization with higher programmability.

2.3 Transactional Memory

Expanding from the concept of transactions in databases, *Transactional Memory* (TM) denotes a mutually exclusive (locked) section of code as an identically-named transaction; but it does not truly enforce it to be mutually exclusive. Instead, it allows multiple threads to enter but execute the code speculatively. Thus, any data objects that must be locked by a given application thread become simpler to manage, as a single transaction can enclose all of the data, instead of requiring low-level locks per item. The straightforward example of a bank account transfer can be continued here, as seen in the left half of Figure 2.2. Instead of locking access to all of the accounts as with coarse-grained locking, the transaction itself will manage its own access to the data needed, namely `acc1` and `acc2` here.

```
         Root              |            Nested
atomic {                   | atomic {
  acc1.withdraw(amount);   |   atomic { acc1.withdraw(amount); }
  acc2.deposit(amount);    |   atomic { acc2.deposit(amount); }
}                          | }
```

Figure 2.2: Root-Level vs. Nested Transactions

Various systems have risen in hardware and software development in order to implement this form of concurrency. To execute speculatively, TM logs transactional operations in-memory via *read-sets* and *write-sets*. Essentially, transactions track the objects that they have read during their run-time along with the written objects and values. These read/write logs are used to ensure operational correctness by detecting conflicts, and they are used most often through two modes of execution: *eager* and *lazy versioning*.

Eager versioning is also known as *undo-logging*, and it does not stray that far from traditional database locking. As transactions read or make updates to items during their execution, they acquire locks on the items eagerly, i.e., at the instant they wish to access the item. If they are unable to acquire a lock, they must *rollback* and undo their operations, retrying at a later point. If transactions succeed in locking an item, they store the current data value into their own write-set and then update the item itself. By instantly performing data writes, the information is already in-place for a transaction to commit and can be read from there if needed. However, the requirement that aborting transactions undo their work is a drawback, as they must iterate their write-set and restore all of the values that were previously valid.

Lazy versioning, on the other hand, leaves everything for a transaction's commit time. Here, transactions perform *redo-logging* and are typically executed optimistically, thus even locking is not performed until commit time. As transactions write data, they do not touch the actual objects themselves, but rather store the new information in their write-set. When they wish to commit, transactions must lock all of the read and written objects, then validate themselves to ensure that the information read is not stale. If any conflicts arise, transactions abort by simply dropping their read- and write-sets. However, if the transactions succeed, they must re-execute their writes by flushing their write-set data to the items.

TM has grown over the years using the above methodologies, sparing developers from the perils of lock-based applications, both in terms of direct programmability as well as composability. The common faults and difficulties of concurrent programming are either prevented entirely or simply managed automatically by a transactional system, lightening the burden of design. *Hardware Transactional Memory* (HTM) picked up from the proposal by Herlihy and Moss [6], while *Software Transactional Memory* (STM) was brought forth by Shavit and Touitou [7]. In HTM systems, transactions are fundamentally limited by memory space and time of execution, along with other problems caused by interrupts, cache misses, etc. While HTM does have the lower overhead of the two, STM does not face those same limitations. Both hardware and software TM systems execute their work in parallel while traditionally satisfying three of the four main database properties: ACI. Note that durability (D) is not supplied if the system is entirely in-memory.

2.3.1 Nested Transactions

Nested transactions originated in the database community through Moss [8] and extended into TM as it was brought into focus. Transactions contained within other transactions are said to be nested, or *sub-transactions*, examples of which are the withdraw and deposit in the right half of Figure 2.2. Nesting was designed to take advantage of the potential for *partial rollback*, i.e., for a transaction to only undo portions of its work if not all of it has been invalidated or conflicted upon. *Flat nesting* is the simplest form of nesting, where every inner (*child*) transaction is executed as part of the top-level (*root*) transaction and does not allow for intermediate executions. Any conflict detected aborts the entire transaction and all of the children, corresponding to the case of basic rollback. Although this method of nesting is the easiest to implement, it serves little purpose, as other methods can truly improve transaction response time and throughput; but to do so, the gains must exceed any overheads introduced by allowing for partial rollbacks.

Closed nesting [8, 9, 10] treats each *parent transaction* as a container for its children. While sub-transactions are executing, they can abort independently from their parent, thereby potentially reducing the scope of rollbacks. When child transactions are finished and validated, they *speculatively commit*, merging their state into the state of their parent. The effects of any children are not seen until the entire root transaction (which has no parent) commits. If a conflict is detected after a child commits, its parent will abort in its entirety, clearing any actions of committed children. Further, a parent transaction can commit only after all the enclosed nested transactions have successfully committed.

Lastly, *open nesting* places an optimistic perspective on sub-transactions. Rather than requiring parent transactions to inherit sub-transaction work (and thus waiting until the root commit before showing the work), nested transactions can commit their changes publicly under the assumption that their parent will commit. This action breaks isolation, similarly to that explored in Chapter 4, but it greatly increases concurrency and performance. To handle transaction aborts, a higher level of work must be implemented, as the transactions must perform operations that explicitly undo their previous functions.

All of the above styles of nesting only allow one sub-transaction to be active at a given time, thus lowering internal concurrency. *Parallel nesting* [11, 12, 13], on the other hand, allows child transactions to execute concurrently with one another, ideally increasing the throughput of the whole transaction. However, parallel nesting raises some issues, with the main problem being that, in addition to already-present conflicts with external transactions, the system now allows conflicts internal to a transaction to occur between its children. Taking this further, because transactions can now have more pieces working in parallel, there are even bigger chances for conflicts between top-level transactions. Methodologies behind parallel nesting are the focus of SPCN in Chapter 3.

Chapter 3

SPCN: Speculative Parallel Closed Nesting

3.1 Introduction

Traditional abstractions for parallel synchronization, such as low-level locks or atomic operations (e.g., *CompareAndSet*), require very precise structure and can cause rather complex issues like deadlock or livelock, especially in large applications. On the other hand, a *transaction* is a general abstraction in which a set of operations are grouped together as one individual, *atomic* action. The property of atomicity ensures that all of the operations in a transaction must be completed or none at all.

Transactions typically keep all of the changes they have made hidden until they finish, or *commit*. A transaction commits if no conflicts have been detected between other transactions and itself. Upon detection of a conflict, such as two transactions writing to the same object, one of the transactions must *abort* and prevent any of its changes from becoming visible. The aborting transaction can either restart from the beginning (*rollback*) and re-execute in its entirety, or, in the case that some of the operations executed are still valid, it can restart from an intermediate point (*partial rollback*).

A lightweight and less intrusive technique that allows for partial rollback is *nesting*, embedding transactions inside of one another. Nested transactions should be compositional, meaning that nested transactions do not perform operations that would break the parent transaction's operational correctness. Traditional parallel code (using explicit synchronization) is not composable, for two functions may conflict over the same synchronized data and then must be re-implemented in order to be used together. *Flat nesting* is the simplest form of nesting, where every inner transaction (*child transaction*) is executed as part of the top-level transaction (*root transaction*) and does not allow for intermediate executions. Any conflict detected aborts the entire transaction and all of the children, corresponding to the case of rollback. Although this method of nesting is the easiest to implement, other methods can improve transaction response time and throughput, but the gains must exceed any overheads introduced by allowing for partial rollbacks.

Closed nesting [8] treats each *parent transaction* as a container for its children. While sub-transactions are executing, they can abort independently from their parent, thereby potentially reducing the scope of rollbacks. When child transactions are finished and validated, they *speculatively commit*, merging their state into the state of their parent. The effects of any children are not seen until the entire root transaction (which has no parent) commits. If a conflict is detected after a child commits, its parent will abort in its entirety, clearing any actions of committed children. Further, a parent transaction can commit only after all the enclosed nested transactions have successfully committed.

Yet, even though closed nesting makes improvements with partial rollback [9, 10], it only allows one sub-transaction to be active at a given time, thus lowering internal concurrency. *Parallel nesting* [11, 12, 13], on the other hand, allows child transactions to execute concurrently with one another, ideally increasing the throughput of the whole transaction. However, parallel nesting raises some issues, with the main problem being that, in addition to already-present conflicts with external transactions, the system now allows conflicts internal to a transaction to occur between its children. Taking this further, because transactions can now have more pieces working in parallel, there are even bigger chances for conflicts between top-level transactions.

To give an example, let us consider the following transactional tree where the parent transaction, T_P, has three closed-nested children: T_{N1}, T_{N2}, T_{N3}. Even though they are part of the same working transaction, T_{N1}, T_{N2}, and T_{N3} can either interact on the same resources (shared or private objects) or access all different objects such that their executions can be considered fully "independent." Here, we assume that T_{N2} conflicts with T_{N1} over an object, thus it must perform the read after T_{N1} has written. On the other hand, if T_{N3} is independent, it can finish early without conflict. Running these sub-transactions in parallel allows the overlapping of their execution, shortening the parent transaction's critical path. Even with conflicts to resolve, the execution is ideally better than, or in the worst case equivalent to, the serial execution.

When applying the idea of parallel nesting to centralized systems, the above notions do not hold as well. Every operation inside of a transaction and its children is on the critical path, thus any overhead added to a transaction processing algorithm will delay the entire execution. On the other hand, in distributed systems, the critical path is mainly the network communication required for requesting data. Thus, adding an overhead for internal processing of transactions does not extensively add to the minimum time required to run the entire transaction. If a transaction's children can be executed in parallel, then their communication in the network is overlapped, and further, their processing and validation is all internal to the node, as they must validate against one another.

Parallelizing transactions within distributed systems is not well-explored, but because distributed systems hold less constraints than centralized systems, it is an intriguing path to pursue. Motivated by the above observations, in this chapter we present SPCN, a speculative approach for executing distributed closed-nested transactions in parallel, capturing conflicts among them at run-time. The problem of activating multiple parts of the same transaction in parallel has already been studied in the past on centralized settings [11, 12, 13]. Unfortunately, for such systems it is not clear how to automate what should be in parallel without asking more of the programmer. Making even the slightest generalization about transactions in a centralized system could result in a decrease in performance if conflicting operations are placed in parallel. Doing so would result in more transactional aborts and, as all of the operations are on the critical path, would slow down the system. For a centralized system to make efficient usage of parallel nesting, the programmer would have to explicitly separate every piece of a transaction into independent sections, which would require extensive work in larger applications.

Instead, we rely on closed nesting for the benefit of partial rollback and to enforce an order of visibility on sub-transactions without requiring the programmer to make them entirely independent. If a programmer has created transactions using closed nesting, the general execution runs through the nested transactions in the order in which they are defined. (Note that the creation of nested transactions is orthogonal to this work, and tools have been developed to aid programmers in creating them [14].) To parallelize such nested transactions, enforcing an order of operation is still necessary if the contents of the transactions are unknown and not independent, as some constraint must be placed in order to process conflicts between the children. However, adding synchronization operations to enforce an order in centralized systems would again add to the critical path of the transactions, creating overheads that outweigh the possible benefits. On distributed systems, these operations, if properly organized, would not add an extensive overhead, as the internal processing of transactions is not on the critical path, thus allowing more innovations.

SPCN provides two different protocols when it comes to the completion of sub-transactions, which are all assigned a *transaction order*, TO. The *Strict* protocol (Section 3.6.1) enforces sub-transaction order directly by keeping *future* children, those with a higher TO, from completing (although they are still activated in parallel) until all of the children prior to them have completed. In SPCN, when a parallel child has completed, we say that a transaction has *sub-committed*. As stated earlier, a nested transaction does not make its actions visible to anything outside of its parent transaction, thus a sub-commit means the same for parallel nested transactions.

The *Relaxed* protocol (Section 3.6.2), on the other hand, allows future children to sub-commit before all of their previous siblings have finished (assuming no conflicts have been found up to that point). However, the protocol allows them to be aborted afterwards if a prior sibling transaction detects a conflict at a later time. This protocol, while allowing a sub-transaction to be aborted after sub-committing, tries to treat the concurrency optimistically. Note that the Strict protocol still requires the conflicting sub-transaction to be aborted, but the Relaxed protocol creates overhead by needing to undo all of the aborted transaction's updates as well as managing additional data to synchronize child transactions and detect internal conflicts.

The above abstractions of nesting can be utilized in any form of transaction or transactional system, such as database transactions or Transactional Memory [7], allowing our algorithm to be widely used. We chose to build SPCN upon a Distributed TM (DTM) system, *Hyflow2* [15], which is implemented in Scala. We evaluated SPCN using an experimental study with the benchmarks Bank, TPC-C [16], STMBench7 [17], and YCSB [18], measuring the performance against closed nesting on up to 20 nodes in Amazon EC2 [19]. Our results reveal that SPCN consistently outperforms the sequential version of closed nesting by as much as 2.85× using Bank, 3.78× with TPC-C, and 1.78× and 2.9× when applied to STMBench7 and YCSB, respectively.

The rest of this chapter is organized as follows. Related work on nesting is detailed in Section 3.2. The preliminary information of the system model, the transactional model, and the management of transactional conflicts is outlined in Sections 3.3, 3.4, and 3.5. The design of SPCN is detailed in Section 3.6, while, finally, experimental studies are reported in Section 3.7.

3.2 Related Work

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [8]. This work focused on the popular two-phase locking protocol and extended it to support nesting. It also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection. One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [9]. They describe the semantics of transactional operations in terms of system states, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible HTM implementations, which work by extending existing cache coherence protocols.

Closed nesting has also been shown as effective in distributed deployments by [10]. In this work authors execute sub-transactions sequentially without parallel activation, thus the cost of the distribution is always on the transaction's critical path. On the contrary, SPCN still exploits the effectiveness of closed nesting but also increases the system's concurrency by overlapping sub-transactions' executions.

Even though Transactional Memory (TM) promises to make concurrent programming easier to the wider programming community, nested transactions are generally not allowed to run in parallel. That is an important obstacle to the central goal of TM. Due to this issue, parallel nesting has been studied in centralized settings [11, 12, 13].

Agrawal et. al. [12] propose XCilk, a runtime-system design supporting transactions that themselves can contain nested parallelism and nested transactions. XCilk shows the first theoretical performance bound on a TM system that supports transactions with nested parallelism. Baek et. al. [11] present NesTM supporting closed-nested parallel transactions. NesTM uses eager version management and word-granularity conflict detections targeting the state and runtime overheads of nested parallel transactions.

Barreto et. al [20] utilize thread-level speculation to execute operations out-of-order, detecting inconsistencies during runtime. The system combines this speculation with Transactional Memory by requiring the programmer to break an application down into coarse-grained transactions, attempting to remove many data dependencies; however, the model assumes that there is no nesting in the transactions. These transactions are separate parts of sequential code and broken down further by using techniques such as compile-time code inspection, thus examining the internals of the transactions. The system uses redo-logs associated by object and also requires any write to immediately lock an object, thus preventing any other parallel transactions from accessing the data. A contention manager is used if separate applications attempt to write the same object, and transactions will either signal the other to abort if it is in the future or will wait until the transaction completes if it is in the past.

Diegues et. al [13] build upon JVSTM [21] by allowing transactions to directly write to Version Boxes (VBox) which hold permanent versions of objects written by top-level transactions. The VBox is extended to allow for tentative versions written by currently-active transactions, meaning that each VBox contains the write-set entry for that object from the running transaction. When sub-transactions complete, all of the relative VBox entries swap ownership to their parent, and once the top-level transaction commits, the latest tentative versions become permanent. The system expects the application to already be broken into separate pieces that have no inherent ordering to them, thus any sub-transaction can commit before any other sub-transaction at any level of the tree, so long as the given application's correctness criteria are not broken.

Differently from the above proposals, SPCN activates transactions in parallel, but it enforces the serialization order as their order of creation, since it does not assume sub-transaction independence. In addition, if a conflict arises, partial rollback can be leveraged for restarting just the executing sub-transaction rather than the whole transaction.

3.3 System Model

We consider a distributed system which consists of a set of nodes. The nodes communicate with each other by message-passing across a communications network. Nodes do not have globally-shared memory, nor do they have a consistent global clock or instance of time, thereby requiring each node to synchronize with the rest of the network as necessary.

Distributed systems may utilize *replication*, meaning that application data will be placed on more than one node as a means for ensuring fault tolerance. However, SPCN is an orthogonal mechanism to replication. It can be deployed in systems where replication is employed, as well as where each node keeps just a partition of the shared resources. In this chapter we focus on such latter systems, allowing us to better showcase the behavior of SPCN without additional overheads due to replication.

3.4 Transaction Execution Model

A set of distributed transactions $DT = \{T_1, T_2, \cdots\}$ is assumed. Transactions share a set of objects $Obj = \{O_1, O_2, \ldots\}$, which are assumed to be distributed (i.e., partitioned) on the nodes of the system. Transactions are modeled as a set of *begin*, *read*, *write*, *commit* and *abort* operations on Obj, and they define a total order in which these operations are executed. Transactions that do not execute a write operation are called read-only transactions; otherwise, they are called update (or equivalently write) transactions. A client requests the execution of a transaction T_i by contacting one node of the system, which is named T_i's originating node.

The transaction processing complies with the control-flow model [22], where objects are immobile and transactional operations are invoked on the owners of the accessed objects. In this model, the nodes responsible for maintaining an object are fixed since the creation of the object and until its deletion. When a transaction performs an operation on an object stored on a remote node, the operation is invoked as a remote procedure call on that node.

The object lookup mechanism relies on a directory distributed across all nodes. A directory performs lookup by using a consistent hashing function on an object's ID. The hash result represents the node in the system that stores the portion of the directory where the ownership of the requested ID is defined. Control-flow-based protocols do not necessarily need the deployment of a directory as just described. For instance, the consistent hashing function can directly return the owner of an object according to its ID. We decide to deploy the directory service, however, because through it we can decide the ownership of objects without letting the consistent hashing function decide. By querying the owner, the object is returned via a message to the requester, or a failure message will be sent if another node has locked the object.

3.5 Managing Contention of Root Transactions

When a transaction T requests an object, it receives a message from the owner containing either the object itself or a signal that the object is locked. If the object is locked, then another transaction is updating the object and a conflict has been detected. In this case, T will abort and rollback (or, according to the closed nesting model, partially rollback if it is a child transaction). If the request is successful, then T stores the object in its read-set along with the object's version, or *timestamp*. Each object in the system has a timestamp that is maintained by the object's owner. Each time a transaction commits a write (or update) to the object, the timestamp for that object is incremented. This versioning is used to detect conflicts during the validation phase of a transaction, thus ensuring that the history observed by the transaction during its execution is consistent.

All transactions buffer their reads and writes into a private, per-transaction, *read-set* and *write-set* during execution. These sets are hash tables organized by using object IDs as the keys. Using a classical lazy approach, write operations are performed locally without interacting with any object owner (if the object has not been already read). The writing transaction, T, stores a pair of $(object, value)$ into its write-set. As said before, at the end of their execution, transactions must perform a validation. The validation is done after locking all the objects written, so that if the validation succeeds, then the updates can be safely applied. To do so, T groups the objects in their write-set by owner and sends messages to the owners to lock the objects. Locking objects on a node is only used during this commit-time as a means to synchronize updates to the objects (i.e., mutual exclusion); no lock is used during the execution of the transactions.

Similarly to when an object is requested, the owners return messages either stating a successful lock or a conflict where an object is already locked. If a conflict is detected, then the requesting transaction aborts and rolls back as before. The committing transaction also sends messages to read all of the latest versions of the objects in its read-set. If any of the objects are locked or have been updated since the transaction read them, meaning that their current version is larger than the version recorded by the transaction at the time of its execution, then the transaction aborts as well.

After successful locking and validation, the transaction increments the timestamp of all objects in its write-set, if it has any, then sends commit messages to the owners of the objects. In the control-flow model, all of the commit messages contain the updated versions of the written objects and the owners will replace the objects in their storage. Afterwards, the owners unlock the objects, thereby allowing other transactions to modify them. At this point, the transaction has committed.

Adding in the concept of nesting, transactions can also have children, allowing for partial rollback. The execution of transactions follow in the same manner as before, but when sub-transactions complete, they simply merge their read-set and write-set with their parent transaction, the transaction directly above them in the hierarchy. In terms of closed nesting, no extra step is necessary, thus no temporary validation is performed nor are any objects queried for or locked by sub-transactions. Only when all sub-transactions in the activation tree are completed will the top-level, or root, transaction begin the full commit, following the same process of locking/validation as above. None of the actions of sub-transactions are visible until the root transaction commits.

What we have described so far implements the well-known Two-Phase Commit (2PC) [23] atomic commitment algorithm, which ensures atomicity on the commit of a transaction.

Even though 2PC is well-known to be blocking upon failure of the coordinator, the issue of how to ensure high availability of the transaction coordinator state is well understood, and a range of orthogonal solutions have been proposed in literature to deal with such failure scenarios (e.g., Paxos Commit [24]).

The concurrency control shown so far is sufficient to guarantee a serializable execution of distributed transactions [23]. We can prove that by relying on other solutions that use the same (well-known) approach. As an example, SCORe [25] adopts the same combination of validating read objects after having locked written objects using 2PC. SCORe is more complex because it provides replicated objects and abort-free read-only transactions. Our proposal allows a smaller subset of the transactional schedules that SCORe allows, thus we can rely on it to claim our correctness criterion, i.e., Serializability.

3.6 SPCN: Speculative Parallel Closed Nesting

In this section we detail our two versions of SPCN, one named *Strict* and the other *Relaxed*. The main distinguishing point between the two is the time at which sub-transactions are allowed to commit. In the former, a sub-transaction cannot commit until the previously-ordered one has already committed. In the latter, we let the sub-transactions commit in any order, but they can be subsequently aborted due to (internal) conflicts with other sub-transactions.

3.6.1 SPCN: Strict Protocol

SPCN begins by initiating a specific number of top-level transactions on a given node, each of which can spawn child transactions and begin executing them fully in parallel. The child transactions are assigned an incremental *transaction order* (TO) to enforce their serialization order. Sub-transactions are not allowed to sub-commit until all the previous siblings have sub-committed, in order to prevent changes from possibly appearing out-of-order. During execution, sub-transactions can internally validate themselves by checking for conflicts with previous siblings (i.e., sub-transactions with lower TO values).

Each transaction (and child transaction) manages its own read-set and write-set to track its own data and changes. When a child sub-commits, as is normally done in closed nesting, the child will merge its read-set and write-set into the parent transaction. Until the merge, however, none of a child transaction's changes are visible to any of the other siblings; otherwise, transaction isolation would be broken and aborts could potentially cascade.

Once a child transaction sub-commits, all of the other children can visibly read its data by observing the parent's data. Note that because of the order assigned to the sub-transactions, children with lower TO cannot observe the data of future siblings. For example, if transaction

T_2 writes to any object, its previous sibling T_1 cannot read any of those changes ever. The Strict protocol accounts for this, as mentioned before, by simply stalling T_2 from sub-committing until T_1 has sub-committed. For this reason we can consider this version of SPCN as more pessimistic than its Relaxed counterpart. However, optimism usually comes with additional overhead, which is often higher than the achievable gain. As we will see in the evaluation, that is mostly the case with the Relaxed protocol, even though there are scenarios where Relaxed outperforms Strict.

Due to the sub-commit order enforced by Strict, the only conflict between siblings that needs to be accounted for is Write-After-Read, where a transaction reads an object, then a sibling with a lower TO writes the object at a later point, invalidating the read. This conflict is detected very easily by finding the intersection of the current transaction's read-set with the parent transaction's write-set, which contains the written objects of all previous siblings. Note that if two sub-transactions write to the same object without reading it, that is perfectly valid as the later transaction cannot publish the write until all of its previous siblings have finished. In most applications, however, it is likely that a transaction has already read an object if it goes on to write to it later.

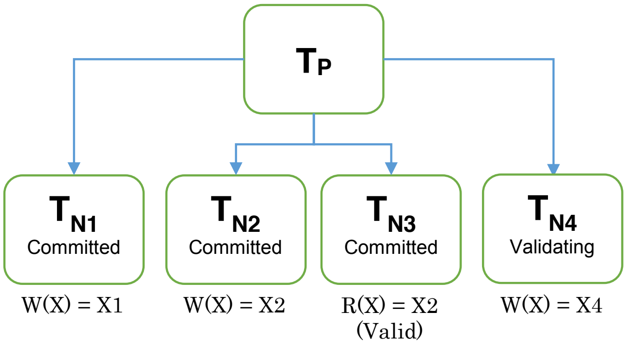


Figure 3.1: Example of SPCN

Because the sub-transactions are only allowed to sub-commit in their order of execution, each root transaction only needs to keep a single version of each object present at any point of time. When a sub-transaction writes to an object, say X, that version is overwritten if a future sibling sub-commits another write to X. An example is shown in Figure 3.1 where three sub-transactions write to the same object. In the Relaxed protocol, there would be three separate versions stored. However, in the Strict protocol, T_{N2} would not be able to publish its writes until T_{N1} had done so, and T_{N4} would similarly wait upon T_{N3}, meaning T_{N2} had already completed. Thus, the object X would be overwritten each time. Doing so removes any need for extra overhead in managing the objects.

Algorithm 1: Strict Protocol

```
1  Procedure ChildBegin
   Input: $TO$, $txnFutures$, $exec$
   Output: None
2  ▷ Get the future of the previous sibling
3  if $TO > 0$ then
4      prevSibling = txnFutures.get($TO - 1$);
5  else
6      prevSibling = $null$;
7  end if
8  ▷ Create a new sub-transaction and run it
9  subTxn = createSubTxn($this$, $TO$, $prevSibling$);
10 txnFutures.set($TO$, subTxn);
11 subTxn.run($exec$);
12 ▷ Increment transaction order
13 $TO$++;

14 Procedure ChildCommit
   Input: $parent$, $prevSibling$, $siblingConflict$, $RS$, $WS$
   Output: None
15 ▷ Wait upon the previous sibling to commit
16 waitUntil(prevSibling);
17 ▷ Check if a previous abort was a sibling conflict
18 if $siblingConflict$ == false then
19     ▷ If not, validate the read-set
20     overlap = intersection($RS$, $parent.WS$);
21     for all objects obj in overlap do
22         if $RS(obj)$ != $parent.WS(obj)$ then
23             siblingConflict = true;
24             abortTxn();
25         end if
26     end for
27 end if
28 ▷ Upon success, merge with the parent
29 mergeRS($parent$);
30 mergeWS($parent$);
```

Algorithm 1 shows the initialization of sub-transactions in the Strict protocol and their commit. In the first function, the parent transaction uses the current transaction order TO and gets the Future of the previous sub-transaction (line 1.4) or assigns null if there was no previous sub-transaction (line 1.6). Futures are an API in the Scala programming language as a means for executing a task and monitoring its completion. The parent transaction then creates the new sub-transaction by giving it a reference to itself as well as its previous sibling, and stores the new Future into the array by order (lines 1.9-10). Lastly, the parent executes the sub-transaction using a dispatcher *exec* and increments TO (lines 1.11-13).

The second function shows the sub-commit of child transactions. At this point, the sub-transaction waits on the immediately-prior sibling's Future to complete (line 1.16). Once the sibling is finished, then the current sub-transaction validates itself. Recall that the only conflict is Write-After-Read, thus the sub-transaction gets the intersection of its read-set along with the parent's write-set, which now contains the writes of all previous siblings (line 1.20). If there is an intersection, it checks to see if any of the read versions are different from the latest versions, and if so, it marks a sibling conflict as having occurred and aborts (lines 1.21-26). Upon successful validation, the sub-transaction simply merges its read-set and write-set with the parent (lines 1.29-30). Note that the $siblingConflict$ variable is a fast-path used to skip this validation (line 1.18). If a sub-transaction aborts because of a previous sibling, then it does not have to backoff or validate again when it comes back to commit. The reason is that the previous siblings are all done, so the sub-transaction can immediately re-read the data that conflicted.

Now, there is a general problem that arises from Strict's simple wait-and-commit methodology. If two sibling transactions are unrelated and do not have any potential conflicts, then stalling the later transaction until the previous sibling sub-commits is not fully effective; but as previously mentioned, independence of the nested transactions is not assumed, as that would have required the programmer to explicitly separate code into completely disjoint sections.

This consideration motivated us to design the Relaxed version of SPCN (Section 3.6.2). However, supporting out-of-order sub-commits requires maintaining multiple versions of the same object, making reads visible, and monitoring a sub-committed transaction until all of its previous siblings are also sub-committed. All of these components are necessary to ensure the correctness of the transactions and they introduce additional overhead, none of which exists in the Strict version; thus, the system's workload should satisfy certain conditions so that the additional overhead's impact is reduced.

3.6.2 SPCN: Relaxed Protocol

The Relaxed protocol is an extension of the Strict protocol that requires more complex structures but also allows for more potential parallelism. Sub-transactions with higher TOs are allowed to sub-commit speculatively even if previous siblings are still active, so long as none of the previous siblings nor the sub-committing transaction found any conflicts up to that point. We name this sub-commit speculative because the serialization order is still not finalized until all previous siblings sub-commit, although the sub-committing transaction, T_C, can publish its changes to the parent before this occurs. Note that, because the sub-transactions still have an order enforced by TO, earlier siblings could still be active and discover a conflict with T_C. In terms of the conflict itself, the only problem is still the Write-After-Read conflict: if T_C read from a variable X and sub-committed before a previous sibling wrote to X.

In order to detect such conflicts with out-of-order sub-commits, multiple versions of objects must be stored and each transaction must store which version of each object it has read. When it comes to keeping track of writes, each object has its own structure, here named as "version tree" ($verTree$), that sorts writes by the order (i.e., TO) associated with the corresponding sub-transaction. The most sensible tree would be an AVL Tree, used for efficient insertion, deletion, and searches, all of which cost $O(log\ n)$ on average where n is the number of levels in the tree.

We also use a hash map ($readHash$) to keep track of transactions reading objects, making the system use *visible reads*, although the reads are only visible after the child has sub-committed. Each object has a hash map where the keys are the TOs representing a version of the object, and the values in each bucket are more TOs representing which transactions read the corresponding version of the object. While adding this structure uses more space per object, it results in faster conflict detection. The tree is better for reading and writing versions based on the TO while the hash is better for knowing which transaction has read a version, as iteratively traversing the trees would take more time.

Child Commit. Algorithm 2 shows the commit function for sub-transactions and the parent thread processing the committing transactions. Starting with the child transactions, they first check if the root transaction is invalid and abort if so (lines 2.3-5). Normally, in a single-threaded transaction, a conflict with already-committed data would cause the whole transaction to abort and restart. However, in this parallel implementation, the children are all different threads and thus the root must set an invalid flag for the children to query. Then, the child waits until the *syncLock* variable is available (line 2.7). The variable is used as a barrier if multiple transactions are to reach this point at the same time.

Once the transaction is able to commit, it must validate its read-set. For each object in the set, it checks the immediately-prior version available in the $verTree$ for the object (line 2.10). If the version of the object is not equivalent to what the transaction has read, or if the version is the same but the data has changed, then the transaction must abort as it is no longer correct (lines 2.11-14). If no conflicts are detected, the transaction makes its reads visible to other transactions in the $readHash$ (lines 2.17-19), then moves on to publish its write-set. Note that the validation of the read-set and the publishing of the reads is separate. The reason is that if transactions published reads as they validated, then they would have to undo all of the publishes if a later conflict was detected, which would be more overhead.

The transaction must observe the $readHash$ entries for each object it is writing and look at siblings which read the immediately-prior version of the object. If any of those readers have a larger TO than the committing transaction, the transaction removes the $readHash$ entry for that invalid sibling and adds it to a list of invalid transactions (line 2.23). For instance, in Figure 3.1, transaction T_{N4} is committing and writing object X which has two versions, X_1 and X_2 corresponding to T_{N1} and T_{N2}. Transaction T_{N4} must only check the readers of X_2 and flag them as invalid if they have TO greater than 4. If T_{N4} sees that T_{N3} has read X_2, it is perfectly valid as T_{N3} is meant to occur before T_{N4}.

Algorithm 2: Relaxed Protocol (Sub-Transaction Commit)

```
1  Procedure ChildCommit
   Input: $RS$, $WS$, $TO$
   Output: None
2  ▷ Check the root's validity before committing
3  if root is invalid then
4      abortTxn();
5  end if
6  ▷ Synchronize with siblings for committing
7  waitUntil(syncLock);
8  ▷ Validate the read-set
9  for all objects obj in RS do
10     check = readPrevious(obj, $TO$);
11     if $check.TO\ != obj.TO$ or $check.data\ != obj.data$ then
12         ▷ The object is out-of-date or has incorrect data
13         abortTxn();
14     end if
15 end for
16 ▷ Publish the visible reads
17 for all objects obj in RS do
18     markAsRead($readHash$, obj, $TO$);
19 end for
20 ▷ Publish the write-set
21 for all objects obj in WS do
22     ▷ Find out which siblings will be invalidated
23     invalidSiblings = getReaders(obj, previousVersion($TO$));
24     for txn in invalidSiblings do
25         clearWrites(txn);
26         invalidSiblings += getReaders(txn.$WS$, txn.$TO$);
27     end for
28     ▷ Update the object
29     writeTree(obj, $TO$);
30 end for
31 ▷ Signal the parent and unlock
32 signal(childCommitted);
33 syncLock.next();
```

For each sibling marked as invalid, the transaction's writes are removed as they are no longer valid (line 2.25). Further, other transactions are marked as invalid if they read any of the objects from the invalid transaction's write-set (line 2.26), as their information is no longer valid as well. The transaction then publishes the object (line 2.29). Once it has completed all writes, the transaction signals that it has committed and then allows the next sibling if it is queued, or simply frees *syncLock* for another sibling's commit (lines 2.32-33).

Algorithm 3: Relaxed Protocol (Parent Transaction Commit)

```
1  Procedure Commit
   Input: invalidSiblings
   Output: None
2  while numCommitted < numChildren do
3      ▷ Wait for a commit and restart invalid children
4      waitUntil(childCommitted);
5      for txn in invalidSiblings do
6          numCommitted−−;
7          restartTxn(txn);
8      end for
9      numCommitted++;
10 end while
11 ▷ Update the read-set with distinct objects from the children
12 updateReadSet(readHash);
13 ▷ Update the write-set with the latest object versions
14 updateWriteSet(verTree);
```

While the conflict detection may seem rather expensive, we optimized the design of the solution so that, in general, it does not slow down transaction response time significantly. In fact, as said above, transactions must only check the immediately-prior version. The reason is that all other possible conflicts would have been detected by other siblings if they occurred.

In the example from Figure 3.1, T_{N2} would detect future siblings that read X_1. Say that some sibling T_{N5} also existed and read X_1. Then if T_{N5} had committed before it, T_{N2} would detect the conflict and would mark T_{N5} as invalid. If T_{N2} had already committed before T_{N5}, then T_{N5} itself would see its incorrect read, even if T_{N4} was not yet finished, as it would notice that it read X_1 instead of X_2. Thus, T_{N4} itself does not need to detect that conflict, and T_{N5} would expectedly read X_4 by the time it finished.

Transaction Commit. The next function shown, in Algorithm 3, is the commit of the transaction which simply handles restarting its child transactions as necessary. While the number of successfully committed children is less than the number of child transactions, the transaction waits until a child has signaled completion (lines 3.2-4).

Upon the commit of a child, the transaction simply runs through all of the sub-transactions marked as invalid, decreasing *numCommitted* and restarting the children (lines 3.5-8). In our implementation, the transactions store a block of code execution when they first begin, thus the restarting of a transaction simply re-executes the block, similar to aborting inside of the transaction while it is running. Once all of the children are successfully committed, the top-level read-set is updated using the $readHash$ across all of the children, and the write-set is updated with the latest versions of written objects (lines 3.11-14). Afterwards, the normal commit of a top-level transaction is executed as described in Section 3.5.

3.7 Evaluation

We built SPCN into the Hyflow2 DTM framework [15], which is a high-performance software infrastructure for implementing distributed synchronization protocols, written in Scala. As a testbed, we utilized *Amazon EC2* [19]. The experiments were run on up to 20 nodes of *c3.8xlarge* machines, where each machine is equipped with Intel Xeon E5-2680 v2 (Ivy Bridge) processors, 32 vCPU, and 60 GB of memory. Each data point is an average of 5 runs.

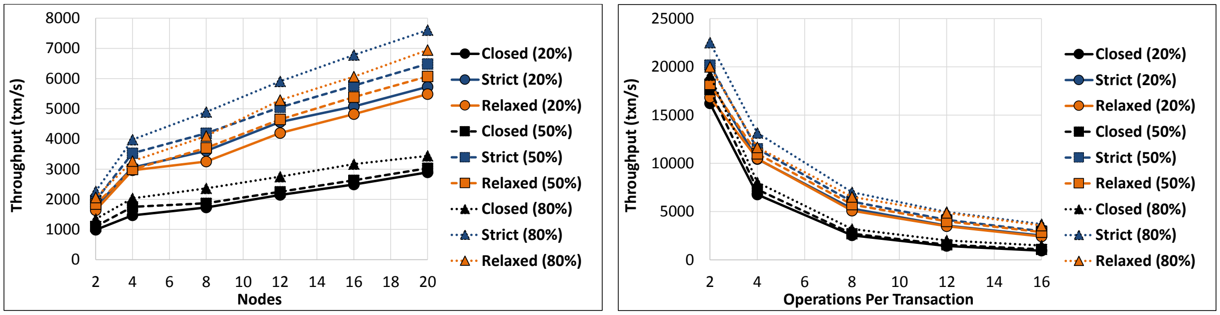
As a competitor, we contrast the performance of SPCN against the sequential implementation of closed nesting, where sub-transactions are executed sequentially without overlap. We cannot contrast our results with any other competitor as none of them support parallel activations of nested transactions. Our deployment is composed of multiple nodes where each runs a number of application threads. Each application thread is an individual client capable of conflicting with any of the other application threads in the system. The number of application threads does not coincide with the actual number of threads processing transactions in the node, because each application thread is in charge of executing one root transaction, and new threads are spawned once a sub-transaction is invoked (thus enabling parallelism).

Four benchmarks were utilized in experimentation: *Bank*, a lightweight benchmark which mimics traditional bank operations; *TPC-C* [16], a popular benchmark that simulates warehouse inventories and on-line processing of item orders; *Distributed STMBench7* [17], an extended version of the heavily-conflicting benchmark built up with a tree of various objects that culminate in graphs at the bottom; and *YCSB* [18], a database benchmark translated to STM by using read and update operations on tables of data.

We performed various tests, such as scaling the size of the system by increasing the number of active nodes; by adjusting the level of contention by varying parameters relevant to each benchmark, such as the number of objects available or the percentage of transactions that are read-only; by allowing internal conflicts between parallel sub-transactions; and by varying the number of operations performed by transactions, as well as the number of active client threads. By doing so, we want to explore multiple configurations in order to understand where SPCN is more effective.

3.7.1 Bank Benchmark

The Bank benchmark has two operations, *balance check*, which opens a set of bank accounts and observes their values, and *transfer*, which withdraws money from one account and deposits it into another. In order to utilize Bank for our testing, we managed to vary the number of operations in order to show the impact of the number of sub-transactions per original transaction. The operations parameter (*ops*) defines how many nested sub-transactions



(a) Transfers per transaction = 8.

(b) Number of nodes = 20.

Figure 3.2: Throughput of Bank varying percentage of read-only operations (500k accounts, 8 application threads per node)

there are. For transfer, each sub-transaction opens two bank accounts and transfers money between them. Thus, if *ops* is 4, then there are 4 sub-transactions, each operating upon 2 bank accounts, thus one root transaction uses 8 accounts.

In addition to *ops* and the number of application threads running on each node, the other parameters are the read percentage indicating how often balance check will occur as opposed to transfer, as well as the number of bank accounts created, which are spread randomly and evenly across the number of nodes running. Changing those parameters means changing the overall contention in the system (e.g., less number of objects means more contention).

Figure 3.2 shows the results of SPCN using the Bank benchmark. In both of the plots reported we configured Bank by deploying 500k total accounts in the system and each node is equipped with 8 threads running the application code. In these experiments we vary the contention in the system by changing the % of read-only transactions in the range of {20, 50, 80}. Figure 3.2(a) plots the total throughput of the system while increasing the number of nodes deployed. As you can see, Strict substantially outperforms closed nesting by as much as 2.25× due to the exploitation of parallel activation of sub-transactions. The improvement increases along with the percentage of the read-only workload because clearly in this case conflicts are reduced and parallelism is more effective. Relaxed SPCN performs closely to Strict, but slightly worse, because all of the sub-transactions are balanced to the same size, thus the extra commit processing holds it down.

In Figure 3.2(b) we fix the number of nodes as 20 and we increase the transactional load by increasing the number of operations made inside a single transaction. The performance clearly decreases while increasing the size of the transaction, but it is interesting to observe how SPCN constantly performs better than sequential closed nesting by as much as 2.72×. The speed-up of SPCN over closed nesting actually increases as the number of operations increases, although the total throughput decreases as the transactions become larger. With more parallel operations, Relaxed's processing does not hold as much weight as it did in the previous experiment, thus its performance is comparable to Strict.

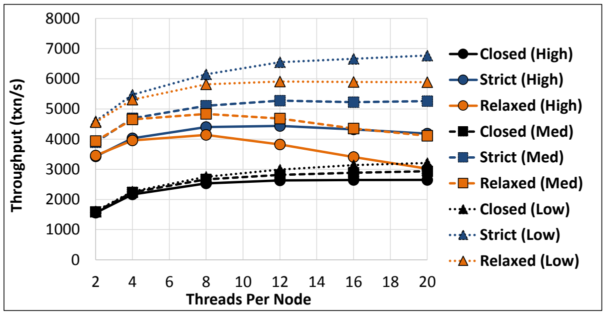
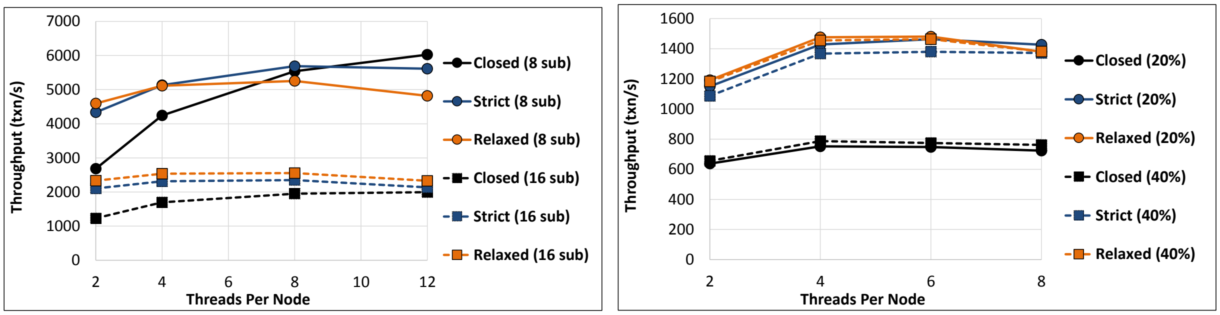


Figure 3.3: Throughput of Bank varying contention levels (8 operations, 20 nodes)



(a) 50% conflict and 8/16 sub-transactions.

(b) 4 sub-transactions with varied conflict chance.

Figure 3.4: Throughput of Bank on 20 nodes (50% reads, 50k accounts, internal conflicts)

In Figure 3.3 we vary the contention level by changing the number of deployed accounts rather than the percentage of read-only transactions, which is now fixed at 50%. We run on 20 nodes and also increase the number of application threads per node up to 20. Each transaction has 8 sub-transaction operations. From the plot it is clear that when the contention is low, thus sub-transactions are likely independent (we recall that accesses to accounts are random) and root transactions do not suffer from several aborts, then the parallel activation pays off. As the number of threads increases, the contention causes Relaxed SPCN to drop in performance, because the processing must be redone multiple times as there are more aborts. The maximum improvement we observed over closed nesting is 2.85×, 2.45×, and 2.2× in the low, medium, and high contention scenarios, respectively.

Lastly, we performed experiments that vary internal conflicts for parallel nesting. If two sub-transactions propagate data between each other (i.e., one writes an item and the other reads that new information), then parallel nesting encounters a conflict. The sub-transactions cannot go fully in parallel as the newly-written data needs the writer to sub-commit. For closed nesting, no overhead is added, as the sub-transactions always go sequentially. In Figure 3.4, we include two experiments with internal conflicts running on 20 nodes, both using 50% read operations and 50k bank accounts.

Figure 3.4(a) shows transactions running with a 50% chance of sub-transactions conflicting.

The number of sub-transactions is changed from 8 to 16 per top-level transaction thread. With 8 sub-transactions, Relaxed starts roughly the same as Strict but performs worse as more transaction threads are added, as Relaxed's conflict processing adds overhead. Closed nesting even manages to overtake both of them in throughput. The reasoning here is that closed nesting essentially performs 4 operations, as 50% conflicts of shared data means that half of the sub-transactions only require re-reading local data. In order to execute in parallel, SPCN has multiple requests (likely at slightly different times) that could be for the same data. At 16 sub-transactions the parallelism pays off. Note that the throughput for each is less than before simply because each transaction is now longer and takes more time to complete. Here, Relaxed is the best. The parallel commits allow the internal conflicts to be processed earlier than they could be in Strict (due to the forced ordering of commits).

Figure 3.4(b) shows a different test with varied conflict chances of {20, 40}%, with 4 sub-transactions that vary in size randomly from 1 to 8 operations. Each sub-transaction has a different computation time, which causes the in-order commit protocol of Strict SPCN to form a bottleneck, as shorter sub-transactions can potentially stall until earlier, longer sub-transactions commit. Relaxed's early processing overcomes that barrier and is also able to process conflicts earlier, allowing for better performance.

3.7.2 TPC-C Benchmark

TPC-C [16] is a larger benchmark simulating stock warehouses with item orders and deliveries. The default breakdown of its five different operations places the creation of new orders and the payment of previous orders as the most frequent. With this profile, 92% of the transactions executed are write transactions. We evaluated by encapsulating any loop iterations as sub-transactions. As for parameters, we change the access locality skew, which is the percentage of transactions accessing the warehouse located on the same node where the transaction is executing. Decreasing this parameter increases the contention in the system because more network communication is required to execute and commit transactions. The benchmark is configured deploying one warehouse per node, thus contention in the system is high, as is usual for TPC-C. In the following plots (except for Figure 3.7) we do not report the performance of Relaxed because it is essentially the same as Strict and we want to avoid showing always overlapping lines.

Figure 3.5 displays results increasing the number of nodes (thus the number of warehouses and threads as well) and changing the transaction access skew from 0% (completely random accesses) to 25% (a slight bias to local accesses). As is shown, Strict is able to scale better than closed nesting, as it increases linearly while closed nesting slowly increases and then decreases again after 16 nodes. The reasoning here is the length of the TPC-C transactions and the high conflict rate. Because many transactions abort very often, and because closed nesting takes significantly longer to perform sets of transactions than Strict does, it is much slower in re-executing aborted transactions. Even though Strict faces high abort levels

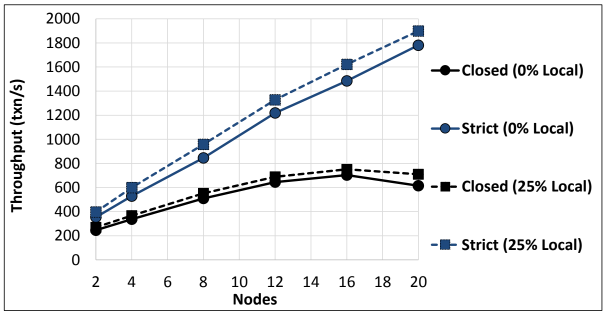


Figure 3.5: TPC-C scalability (8 application threads per node)

(even higher than closed nesting), its parallelized transactions can re-execute much more quickly. In terms of locality, the scaling trends remain the same for both closed nesting and Strict, although 25% locality results in higher throughput for both of them. The largest performance increase over closed nesting in this chart is 2.9× and 2.68× for 0% and 25% local skew, respectively.

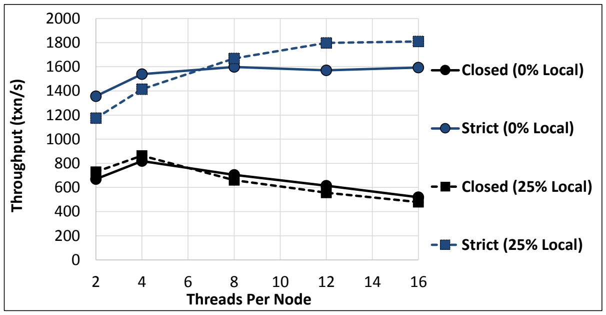


Figure 3.6: TPC-C throughput with increasing contention and varying locality (20 nodes)

The results plotted in Figure 3.6 show the behavior of the two competitors while varying the application threads per node and the access skew, and we fix the system at 20 nodes. As is shown, at 0% local skew, Strict levels out due to the very high contention and high network communication from the majority of transactions often being external. Yet, Strict is able to sustain as the number of clients increases up to 320 total, while closed nesting decreases in performance, unable to handle the conflicts. Closed nesting decreases as well for 25% local skew, while Strict increases and performs better than itself at 0% local skew, leveling out at 12 threads per node. The reasoning here is that, with the local bias, transactions are less often external, thus Strict can re-execute aborted transactions even faster if they are local. The speed-up over closed nesting increases with the number of threads due to increasing contention and closed nesting decreasing in performance, with the maximum being 3.07× and 3.78× for 0% and 25% local skew, respectively.

Figure 3.7 demonstrates TPC-C configured with all read-only transactions. Here, there is no possibility of conflict because no write is invoked. In the plot we report the performance of having 4 and 8 sub-transactions per original transaction. Clearly, Strict and Relaxed SPCN

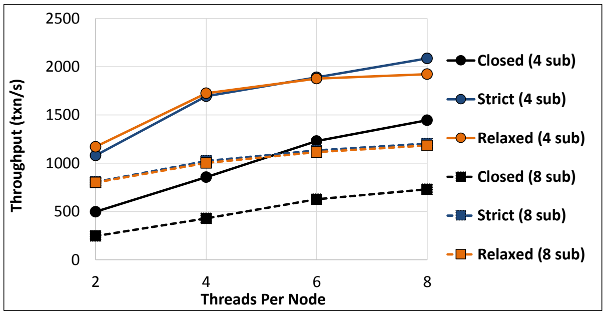
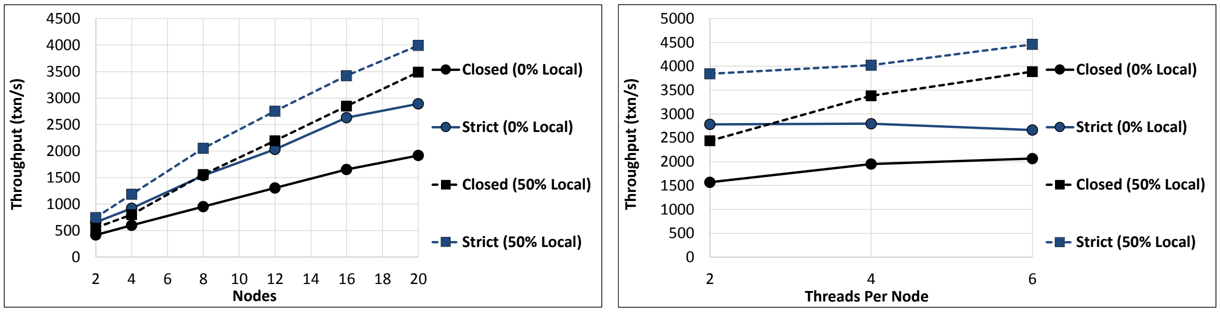


Figure 3.7: Throughput of TPC-C with read-only transactions (10 nodes)

are nearly the same without potential write conflicts. Relaxed slightly worsens with more root threads per node because of its processing, thus slowing commits down as there are more threads than the cores can typically handle. The best speedup is 2.35× and 3.28× for 4 and 8 sub-transactions, respectively.

3.7.3 STMBench7 Benchmark

STMBench7 [17] is a structural benchmark which contains a multi-level tree containing up to 7 different types of objects in a predefined hierarchy. At the bottom of the tree is a graph of objects called *AtomicParts* which are randomly interconnected at the structure's initialization. Functionally, there are *traversals*, *structural modifications*, and general *operations*. Traversals and structural modifications are traditionally disabled because they extensively search and modify much of the structure, often spanning the whole tree. For the operations, there is still a chance for conflict, as different operations will use different sections of the tree and could potentially end up at the same *CompositePart* object, for instance, which are objects at the next-to-last level of the tree.



(a) Application threads per node = 4.

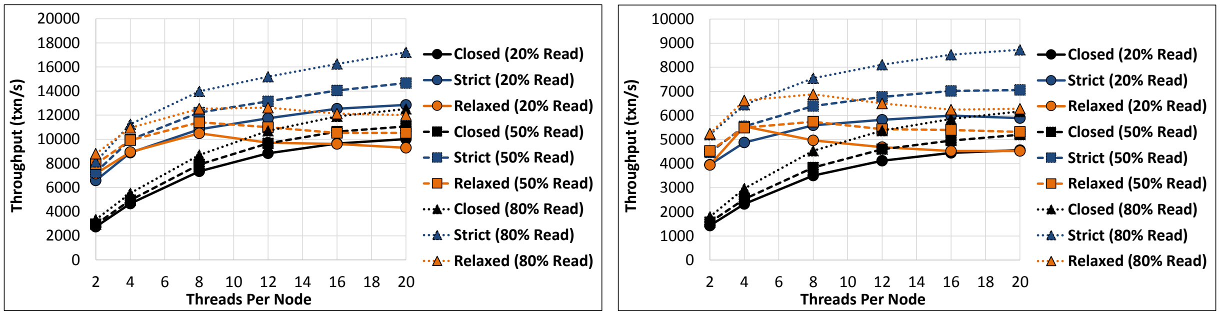
(b) Number of nodes = 20.

Figure 3.8: STMBench7 throughput varying locality and 20% read-only transactions

In the reported experiments we managed to vary the percentage of read-only transactions (i.e., 20% and 50%) and the percentage of transactions accessing local objects rather than objects chosen randomly (i.e., 0% and 50% of skewed accesses). Similarly to TPC-C and warehouses, each node is allocated one tree structure, and each tree has 4 levels of *Assembly* objects. Because of this data organization, it is better to limit the number of threads executing in parallel on one tree; otherwise, the level of conflicts becomes too high. As a result, we run STMBench7 with up to 6 threads per node. Lastly, we left Relaxed out of the charts as the complexity of the data structure added too much conflicting overhead. The performance was greatly lower and we did not wish to affect the scale of the other data. Overall, we can conclude that STMBench7 produces a complex structural workload that is not suited for Relaxed SPCN.

In Figure 3.8(a), we show the behavior as the system is scaled. Each node executes 4 threads, with 20% read-only transactions, which also means higher contention. We increase the number of nodes from 2 to 20, while we also vary the local skew between 0% and 50%. As is shown, higher local skew allows both closed nesting and Strict to perform better due to less need for external communication; however, Strict still outperforms closed nesting at both localities, with the best result being 1.61× closed nesting's performance.

The results in Figure 3.8(b) show the operation at 20 nodes with 20% read-only transactions, as we vary the number of threads per node and the local skew. As explained earlier, the contention in this benchmark is very heavy, thus the performance of Strict at 0% local skew actually degrades from the beginning. On the other hand, closed nesting increases slightly but starts to level off. The reasoning for this behavior is that the 0% local skew requires external communication for the majority of transactions, and the parallelism within Strict has many sub-transactions executing all at once resulting in higher conflict than closed nesting. At 50% local skew, Strict manages to increase, however, and still manages to have higher throughput than closed nesting. Overall, the highest throughput increase here is 1.78×.



(a) Low Contention (100 rows, 10 accesses). (b) High Contention (50 rows, 20 accesses).

Figure 3.9: Throughput of YCSB varying contention and read-only percentage (20 nodes)

3.7.4 YCSB Benchmark

The YCSB [18] benchmark originated as a means of testing database information with SQL-style operations. For testing SPCN, the benchmark simply represents tables of information with different ways to manage the setup and contention possibilities. The main parameters used for the benchmark are the number of rows within a table, the number of cells within a row (or number of columns), the number of cells that a transaction accesses or operates upon, as well as the locality skew and the percentage of read-only transactions. Note that in all of the below experiments, we kept the locality skew at 0%, thus random and more often external requests are made.

One table is allocated per node, and there are two main operations, read or update. Transactions begin by choosing a table from a node, then by choosing a row in the table. The number of rows strongly influences contention, as does the number of accesses performed by a single transaction. For low contention across 20 nodes, we gave each table 100 rows and 100 cells per row, with transactions accessing 10 cells at a time. For high contention, we cut the number of rows in half to 50, kept the cells at 100, and doubled the number of cells accessed to 20 per transaction.

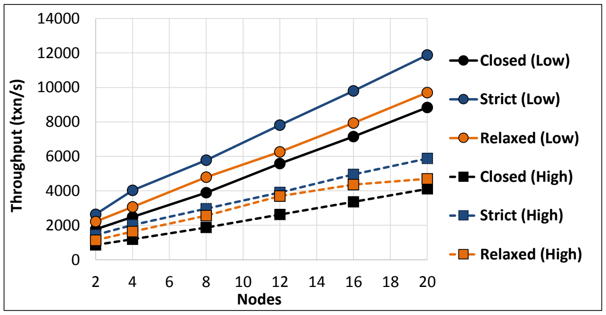


Figure 3.10: YCSB scalability with 12 threads per node and 20% read-only workload

In Figure 3.9, we hold the number of nodes in the system at 20 and vary the number of threads executing per node, the read % of transactions, and the contention level. At both contention levels, closed nesting and Strict follow similar trends, although the average throughput at high contention is roughly half the throughput at low contention. Further, Strict remains better than closed nesting at the vast majority of settings. Notice that one line of closed nesting and one line of Strict intersect, but they are at different ends of the configuration spectrum. In both charts, the bottom Strict line is at 20% read transactions and intersects with the top closed nesting line at a high number of threads. But that closed nesting line is at 80% read transactions, meaning that closed nesting only reaches similar performance to Strict when Strict has 4× the number of write transactions, therefore much more conflict. The best speed-up for Strict here is 2.46× for low contention and 2.9× for high contention. On the other hand, Relaxed cannot deal with the growing contention across 20 nodes, although it begins by outperforming Strict as seen in Figure 3.9(b) using 2 and 4 threads.

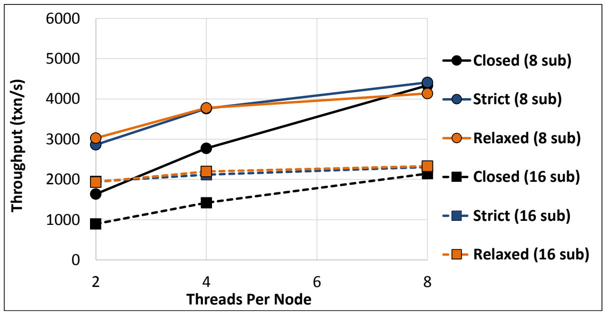


Figure 3.11: YCSB throughput with 50% conflict and 8/16 sub-transactions

In Figure 3.10, we keep the number of threads constant at 12 per node, as well as using 20% read-only transactions (thus, more conflict), while we vary the contention level again and the number of nodes in the system. In this experiment, all competitors scale linearly with the number of nodes. Across the two contention levels, the maximum performance gain for SPCN is 1.69×.

Figure 3.11 shows transactions running with a 50% chance of internal sub-transactions conflicting. The number of sub-transactions is changed from 8 to 16 per top-level transaction thread. Because all of the sub-transactions are the same size, we observed results similar to the conflicting test using Bank as shown in Figure 3.4(a).

Chapter 4

AsR: As-Serializable Transactions

4.1 Introduction

Transactions are synchronization primitives that allow an application to execute operations on shared objects as singular, *atomic* invocations. Database management systems (DBMSs) represent the de-facto standard to provide applications with transactional support. In doing so, they rely on concurrency controls that implement specific rules for satisfying the data consistency requested by the application. We can consider *Serializability* [23] as the gold standard consistency level, because it allows application programmers to focus entirely on developing their program's business logic, without needing to determine any *anomalies* that may occur from concurrent data usage. We can intuitively define an anomaly as a non-desirable interleaving of operations that produces inconsistent results for an application. However, inconsistency depends entirely upon the creator's expectations for their application; for instance, some applications may allow their users to see stale data without any problems, while others require their users to see the most up-to-date information possible.

After a qualitative evaluation of state-of-the-art, production-level DBMSs (e.g., [26, 27]) that provide strong, consistent concurrency controls, it can be summarized that the way anomalies are prevented can be too *conservative* given the highly parallel commodity hardware platforms currently available. As evidence for the generally conservative controls, the common technique to prevent the invalidation of accessed objects is to simply lock them.

Consider a database where a transaction T_R is reading an object X. As an effect of this read operation, X is (shared) locked until the commit of T_R. However, if another transaction T_W wishes only to write X concurrently (and after the read operation of T_R has finished), it should be able to do so according to the Serializability specification (i.e., T_W is serialized after T_R). Unfortunately, it is unable to perform the operation due to the presence of the lock itself. Clearly, the lock on X prevents the development of other undesirable scenarios that would corrupt T_R's execution (see Section 4.5 for more details), but in this specific case the abort of T_W could be avoided.

To address potential disadvantages of these controls, one could think of increasing optimism by allowing transactions to proceed and detect inconsistencies in another manner, rather than operating eagerly and locking others out (as most DBMSs do). This solution requires a new concurrency control design and additional overheads, which are not necessarily desirable. Here, an altered transaction processing methodology is proposed that commits Serializable transactions by leveraging the weaker isolation levels defined in the ANSI/ISO SQL specification, which are already provided by most well-known DBMSs (e.g., Read Committed [28]). This way, on the one hand, we preserve the transparency with respect to the programmer, as they do not need to be aware of the new processing technique given that committed transactions are still serializable; and, on the other hand, the overall performance can benefit from handling transactions using concurrency rules designed for weaker isolation levels.

To minimize the re-engineering of typical DBMS systems, the focus lies on the hierarchy of isolation levels reviewed in Section 4.5. In the new processing scheme, transactions will execute optimistically at the lowest isolation level, utilizing additional data to determine inconsistent executions. If a transaction aborts repeatedly when running under weaker isolation levels, they may upgrade to more tightly-ruled levels (and could eventually be committed using Serializability if necessary).

As-Serializable (AsR) transactions are based on the idea that any transactional read operation should leave an identification marker for itself on each accessed object (a technique known as a *visible read*). Any subsequent transactions that intend to write on a marked object will observe the readers and *invalidate* them, by placing their own identification marker within each reader's meta-data. At commit-time, transactions will observe those invalidating transactions, and if any have committed, then the current transaction must abort. While this approach may still falsely abort transactions that should be able to commit, it does ensure that non-serializable transactions may not commit, and in fact allows for better processing than simply locking other transactions out. Further, to ensure guaranteed correctness with no false negatives, fully-connected graph processing would be required to track all dependencies between transactions, thus causing the expenses to outweigh the benefits [29]. Hence, this slightly-pessimistic processing does not add much overhead and simultaneously allows for more transactions to complete.

Accordingly, visible transactional reads should be applied to any isolation level weaker than Serializability in order to preserve the overall guarantee that transactions are only committed as if they were executed under Serializability. Given that, once a transaction is activated under a weaker isolation level (e.g., Read Committed), the DBMS concurrency control still operates as originally designed (e.g., locks are acquired on written objects) and processing code is performed before accessing an object. As shown later in the evaluation study (Section 4.8), the new processing scheme is able to commit transactions mostly under isolation levels weaker than Serializability, thus increasing the overall concurrency level and, further, performance.

The design choice of exploiting visible read operations copes well with eager DBMS concurrency controls where locks are acquired on the accessed objects for both reading (shared) and writing (exclusive) operations. This fact implies that meta-data are already modified by the default logic itself, thus the overhead of implementing visible reads by updating some additional meta-data is alleviated. In addition to that, processing is performed before letting the concurrency control access the database index to retrieve the actual object. As a result, if a transaction should not access some object, it is aborted without fetching the actual data. That reduces the contention on the index, thus allowing other transactions to proceed faster.

Given its design, AsR is very effective when the workload shows at least a moderate contention level. In fact, when transactions are mostly non-conflicting, there is no actual contention on the meta-data used (e.g., locks), and thus the benefits of AsR with respect to the original concurrency control are reduced. Considering the extreme case of transactions accessing disjoint parts of the database, they could (in principle) run without locking the accessed objects. In such a scenario, there is no execution schedule that AsR allows but the original concurrency control does not. However, as shown in the evaluation study, the limited overhead of AsR enables the achievement of similar performance to the original concurrency control in such scenarios. That makes AsR a concrete alternative to Serializability in both favorable (with contention) and "adverse" (without contention) scenarios.

Berkeley DB Java [26], the well-known and widely used open-source DBMS, was utilized as the basis for AsR, with the new processing added to the transactional operations. The system can be easily configured to run under AsR rather than Serializability, which means that programmers can execute all existing applications using this altered Berkeley DB version without any modification to the application source code, counting upon the same guarantees provided by the native Serializable concurrency control. To evaluate AsR, two well-known database benchmarks (i.e., TPC-C [16] and TPC-W [30]) and one synthetic benchmark (i.e., Bank) were used. Results confirmed the benefits in high-contention scenarios and demonstrated that AsR provided an improvement in throughput of multiple orders of magnitude; although the explicit boost is quantitatively irrelevant, as the DBMS itself simply cannot make progress in these settings.

The rest of this chapter is laid out as follows. Section 4.2 begins by covering related work dealing with transactions and consistency. In Section 4.3, general API expectations and assumptions about DBMSs are explained. Section 4.4 provides an overview of AsR's transaction processing via an example transaction history. Section 4.5 covers the background of isolation levels and the various anomalies that can occur to violate application correctness. Into the implementation, Sections 4.6 and 4.7 describe AsR's lower-level details and the meta-data processing. The two sections are separated via a specific type of anomaly in order to create a step-by-step understanding. Lastly, Section 4.8 details the evaluation results of AsR from experiments run against Berkeley DB's default isolation settings.

4.2 Related Work

The general concepts of transactional consistency in databases, along with tools such as predicate locking, were explored early in [31]. In it, Eswaran et. al. define the consistency of a database, scheduling of transactions, the phenomenon of "phantoms" (i.e., phantom reads), the method of locking logical sets of data as opposed to explicit items, among other properties and notions.

Berenson et. al. [28] explore the main isolation levels defined by the ANSI/ISO SQL standard and perform a critique of them, arguing that the specification leaves out many expected forms of operation and that the standard is not hardly defined. The critique reinforces explicit meanings for the isolation levels (as is also explored in Section 4.5) and also defines a new level known as *Snapshot Isolation*, which holds lesser consistency than *Serializability* but can allow for better performance. This form of isolation is so-named as transactions are given "snapshots" of the transactional system at the time they start, i.e., data will remain the same from a transaction's perspective for its entire lifetime. Thus, only concurrent write-write conflicts will cause transactions to invalidate.

Snapshot isolation is further explored in some works which attempt to adapt it towards Serializability through alternative means. However, many require extensive dependency processing and possible modification of the transactional applications being operated [32, 33, 34], adding significant overhead.

Relaxing the consistency requirements of application transactions has also been researched. In particular, Li et. al. [35] propose the notion of *RedBlue consistency*, an alternative method used to serialize transactions in a replicated system. Blue transactions are capable of being performed lazily (in the manner of eventual consistency), while red transactions must be explicitly serialized with one another and require cross-node communication. Blue operations can lighten the processing overhead of the system, and thus an infrastructure named Gemini was created to coordinate red and blue transactions. However, the proper setup relies upon the application developer, requiring them to analyze their application and separate all of its functionality into the respective operation types. To relieve some of this burden, Sieve [36] attempts to automate this consistency-level allocation by performing static and runtime analysis of an application's operations, although misclassification errors can still occur and require the developers to settle them.

Terry et. al. [37] utilize *Service Level Agreements* (SLA) for cloud database systems, circumventing lower performance by allowing relaxed system rules for limited periods of time. Developers can declare an SLA for Pileus, the database implemented in the paper, that ranks different consistency guarantees vs. the latency experienced in the system. If a given application request is expected to exceed defined latency periods (due to more recent information existing on nodes much farther away), then the consistency can be dropped to whatever is defined by the application creators, potentially improving the performance. Meta-data processing is used by internal algorithms to provide conservative estimates of how up-to-date a given node in the cloud system is, which are then used to determine where to send application requests. The application is notified of each request's consistency as they are returned, thereby allowing it to take different action as necessary (although this capability requires programs to be developed around these different execution possibilities).

4.3 Assumptions and API

AsR finds its best deployment in a multi-core hardware platform where threads can execute in parallel, thus enabling the possibility of having a high concurrency level. In the current prototype, the DBMS processes transactions in-memory without logging its operations on a stable storage. This execution mode makes the database not resilient to machine failures, as the data are not durable; however, this decision is not related to the design by any means. The database can still be set to use logging, and if the administrator accepts restoring the database from the last committed operation (and not mid-operation), then AsR applies just the same. By restoring from the last committed operation, it is not necessary to log AsR's meta-data. Non-durable execution was simply selected for clearer results and better understanding. In fact, AsR is a solution for speeding up the transaction processing and it can be integrated on top of appropriate existing techniques for providing database durability.

AsR is meant for DBMSs that expose (at least) the following APIs to the application (where the below syntax matches that of Berkeley DB, the prototype DBMS):

- `put(X,V)`: a write operation on an object with primary key X to store value V. If X does not exist in the database, it is inserted;
- `get(X)`: a read operation that fetches an object with primary key X from the database. If X does not exist in the database, it returns an empty set;
- `entities(X,Y)`: the operation performs a query on the database to retrieve a range of objects with keys between X and Y;
- `begin`: the operation that begins a transaction. The programmer can specify the isolation level manually through this call or via the database. In the prototype, AsR is configured as a platform parameter and it replaces Serializability to allow transparent compliance with existing transactional applications;
- `commit`: the operation that finalizes the transaction's work;
- `abort`: the operation that intentionally aborts a transaction (which is different from the abort triggered by the concurrency control).

It is also assumed that the DBMS provides an eager concurrency control where shared and exclusive locks are acquired when read or write operations are performed respectively (i.e., encounter-time locking). This assumption complies with the implementations of concurrency controls integrated with several existing DBMSs [27, 38]. The processing scheme proposed by AsR can, in principle, also be deployed on top of optimistic concurrency controls (e.g., those that acquire only exclusive locks before entering the commit phase), but the design choices would have less effect.

Lastly, AsR has been designed assuming the four isolation levels specified in the ANSI/ISO SQL specification [28] (i.e., Read Uncommitted, Read Committed, Repeatable Read, Serializability), which will be detailed in the next section.

4.4 Transaction Processing Overview

The core idea of AsR is to transform the life-cycle of a single transaction from executing under one isolation level L (i.e., the one specified by the application) to executing by leveraging multiple isolation levels. We select the weakest isolation level that disallows any dirty read, namely *Read Committed* (RC), and let transactions execute as they would at that level, albeit with extra processing that is responsible for identifying executions that may violate Serializability (so called "dangerous" transaction histories). When a transaction is repeatedly aborted because it continuously detects these dangers, it is *upgraded* to the subsequent isolation level, which is *Repeatable Read* (RR). The processing also protects the execution of RR transactions due to their possibility of breaking Serializability (but only in the case of having phantom reads, as will be explained shortly). The upgrade process also occurs for RR transactions that abort more than a configured threshold, thus upgrading them to *Serializability* (SR).

This processing scheme may seem inefficient because it lets transactions possibly abort (and upgrade) more than the original concurrency control, which does not pay any upgrade cost, as well as any additional processing overhead. However, transactions running under lower isolation levels are much faster than those under SR, which gives us the flexibility to abort and restart more often. In addition to that, they are less conservative, thus the level of parallelism (and therefore performance) increases. If the workload allows transactions to commit after a limited number of retrials and without being upgraded (especially up to SR), then the proposed processing scheme becomes very effective, as transactions satisfy SR without actually being held by its constraints.

To detect and prevent anomalies, additional meta-data (external to the database's actual information) are used, where each piece of meta-data has a constant size that is independent of the database objects' sizes or the transactions' sizes. To give an example of what meta-data is collected, Figure 4.1 shows a transaction history that is not serializable.

In the figure, there are three transactions (T_1, T_2, and T_3), which share certain objects in their executions. The object meta-data is shown at two different points in the execution history, along with the transactional meta-data, in Figure 4.2. The object meta-data holds the latest transactions that read the given object, as well as the last transaction to write the object.

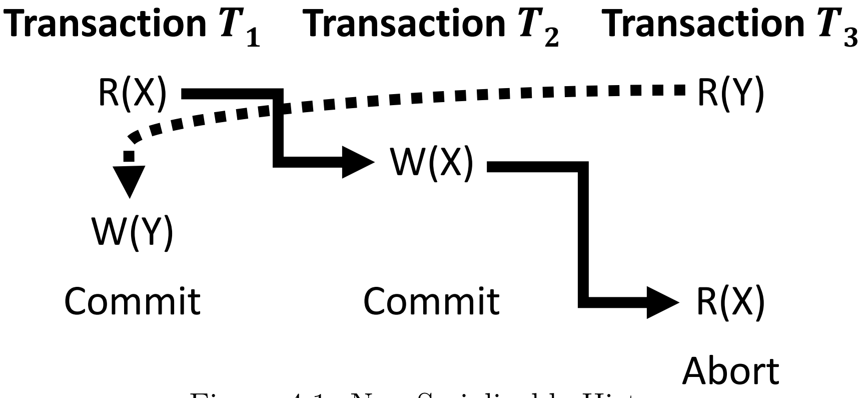


Figure 4.1: Non-Serializable History

Transactions T_1 and T_3 first perform reads on objects X and Y, respectively, making their reads visible as in Figure 4.2(a); more specifically, T_1 occupies the first position for readers of X and T_3 occupies the third position for readers of Y. Then, due to the order of operation, it is simple to observe that T_1 must read a different value of X than that which T_2 writes, thus creating a Read-Write (RW) anti-dependency. Logically, as transactions are atomic, we see that T_1 must occur before T_2 ($T_1 \prec_X T_2$).

When writing, T_2 sees that anti-dependency via the visible read on X's meta-data. Now, in order to continue with its normal write operation, T_2 must perform two extra processing steps. Firstly, it must mark itself as the latest writer of X, as seen in Figure 4.2(b). Secondly, it must look at all of the readers of X at the current point in time (only T_1 here), then mark itself within each reader's meta-data. In this example, T_2 marks itself within T_1's data, as seen in Figure 4.2(c), to denote the anti-dependency. Similarly, transaction T_1 goes through the same steps with object Y and its anti-dependency with T_3 ($T_3 \prec_Y T_1$). Then, with their writes performed, T_1 and T_2 can commit.

Lastly, we come to a Write-Read (WR) dependency between T_2 and T_3 on X ($T_2 \prec_X T_3$). Using all of these dependencies, a chain can be created to find what must be the serialized order of transactions, which in this example is $\{T_3 \prec_Y T_1 \prec_X T_2 \prec_X T_3\}$. However, if such an order contains a *cycle* (i.e., a transaction is found at multiple places in the chain), the operations are not serializable unless the cycles are removed. In this example, T_3 cannot be serialized with the others as it must simultaneously occur before T_1 (because of Y) and after T_2, hence indirectly after T_1 (because of X). With the meta-data, when T_3 goes to commit, it can observe that T_1 should occur after itself (T_3) has committed; but, as T_1 is committed at this point, it leaves a dangerous possibility that inconsistent data has been used. Thus, T_3 must simply abort.

Now, we can compare the above execution to the one allowed by the eager DBMS controls. By the locks acquired during reads, T_2 will abort as it cannot write X, as T_1 has a read-lock on it. With T_2 gone, the entire cycle does not occur, thus the other two transactions should

	Readers			Writer
X	T_1			∅
Y			T_3	∅

(a) Object Meta-Data Before T_2 Writes

	Readers			Writer
X	T_1			T_2
Y			T_3	T_1

(b) Object Meta-Data After T_1 Writes

	InvWriters		
$\boldsymbol{T_1}$		T_2	
$\boldsymbol{T_2}$			
$\boldsymbol{T_3}$	T_1		

(c) Transaction Meta-Data

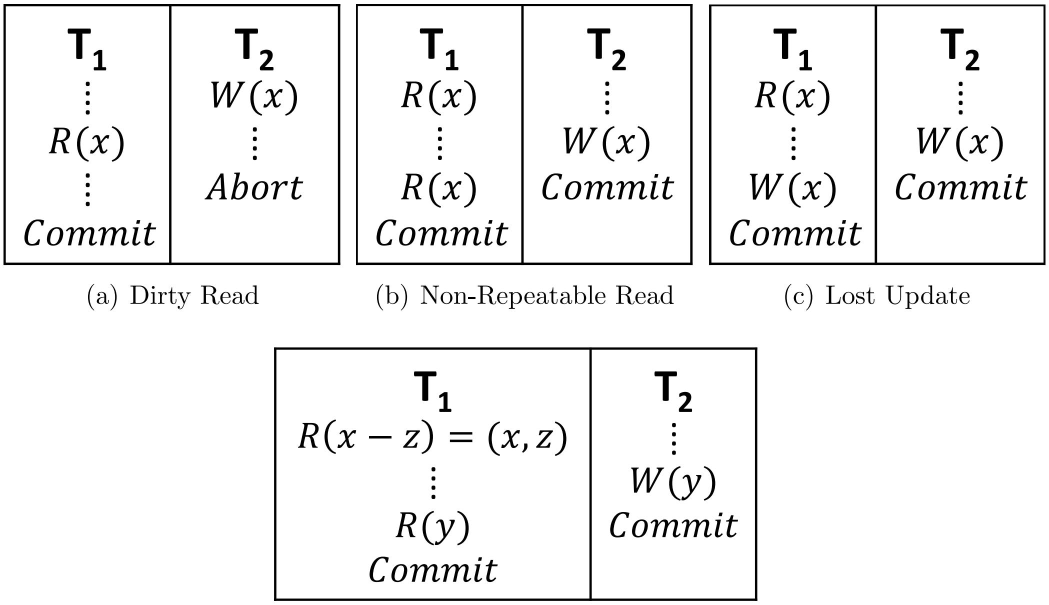
Figure 4.2: Meta-Data for Figure 4.1 Example

be correct and commit; however, that is still not possible. Because T_3 holds a read-lock on Y, transaction T_1 will also abort. Thus, only T_3 will complete, which is the inverse of the previous execution, where T_3 is the only one to abort.

In the next section, the general isolation levels from the ANSI/ISO SQL standard [28] are explained, along with possible problems that can occur in each level, potentially invalidating an application's execution.

4.5 Anomalies, Isolation Levels, and Eager Concurrency Control

Consistency and *isolation* are closely-related properties of database systems. The former is more abstract, defining that transactions must always move the system from one valid state to another. Validity is reliant upon the application that is executing, although a broad



T_1	T_2
⋮	$W(x)$
$R(x)$	⋮
⋮	$Abort$
$Commit$	

(a) Dirty Read

T_1	T_2
$R(x)$	⋮
⋮	$W(x)$
$R(x)$	$Commit$
$Commit$	

(b) Non-Repeatable Read

T_1	T_2
$R(x)$	⋮
⋮	$W(x)$
$W(x)$	$Commit$
$Commit$	

(c) Lost Update

T_1	T_2
$R(x-z) = (x,z)$	⋮
⋮	$W(y)$
$R(y)$	$Commit$
$Commit$	

(d) Phantom Read

Figure 4.3: Transaction Read Anomalies

example of consistency is the requirement that transactions starting in the future, relative to previously-committed transactions, should see the effects of those older transactions.

The isolation level of a system actually influences the definition of consistency for any given application. Isolation defines the level of visibility for transactions' actions, i.e., when their outcomes can be seen. In particular, there are four general levels defined as the standard for databases, although they are not the only possible settings. To best understand them, let us start from the bottom and move upwards, observing the fundamental anomalies. Note that any anomaly mentioned for a given isolation level also exists for all prior levels.

4.5.1 Dirty Read

Read uncommitted isolation is the most basic level. Essentially, there are no constraints on the operations of the transactions. Their view of the system does not have to remain consistent throughout their lifetime, nor does it even have to be valid by the time that they commit. No explicit synchronization must be done between any transactions. This isolation level is the only one which allows *dirty reads*, reading data that is not confirmed as permanent.

Figure 4.3(a) shows an example of such an anomaly. Transaction T_2 writes to an object X, then T_1 reads X. No conflicts occur because read uncommitted does not place any locks or synchronization on objects (except during the actual writing of the data). Thus, T_2 can abort while T_1 goes on to commit. T_1 operates using invalid data for X, and in a typical application, such a chain of events would lead to incorrect execution. Note that the write could also be an insert, meaning that X does not exist; if that is the case, then T_1 operates using data that is not actually there.

Read committed isolation eliminates dirty reads by forcing transactions to keep *write-locks* on modified objects until the transactions commit or abort. Transactions must check these locks when reading or writing an object, thus, in the previous example, T_1 would not be able to read X until T_2 finishes and releases the lock.

4.5.2 Non-Repeatable Read and Lost Update

Now, the next anomaly to correct exists in two forms. A *non-repeatable read* is seen in Figure 4.3(b). The anomaly is exactly what the name says: transactions may not always be able to read the same object multiple times. In the example, T_1 reads X both before and after T_2 writes X and commits. Note that both of the $R(X)$ statements could be direct reads or also reads that cross a range of values that intersect with X. Here, T_1 is seeing an inconsistent view of the system, as it has different values for X at different points in its lifetime, thus breaking the idea that the transaction is atomic.

Similarly, in Figure 4.3(c), T_1 actually writes X instead of reading it another time. It is using out-of-date information to perform other operations and then to update that object, making it as if T_2's write to X never occurred, thus creating what is known as a *lost update*.

Those two sides of the same anomaly are corrected by the *repeatable read* isolation level. The correction is to simply add more locks, this time on the objects being read. Thus, when a transaction reads an object, it holds a *read-lock* on the object until it commits or aborts. In terms of concurrency, multiple transactions can hold read-locks on the same object, sharing it. However, if a transaction wishes to write an object that other transactions hold a read-lock on, it cannot. Hence, neither of T_2's writes can be performed in the previous examples until T_1 has finished with X.

4.5.3 Phantom Read

The last anomaly to understand is the *phantom read*. Without extra synchronization, it is possible for an item to suddenly appear in the middle of a transaction's lifetime. A transaction can attempt to read an object directly or through a *range query*, but if the value does not exist, simply nothing is returned. Despite the fact that the read failed, the transaction can still retrieve the item in a later read, if it has been inserted and committed by another transaction.

As seen in Figure 4.3(d), transaction T_1 views a range from X to Z but only receives those two values in particular, with no object Y existing. Then, another transaction T_2 inserts Y into the system. Now, T_1 can perform another read that could include Y within it, and it will indeed return Y. Note that both of the reads could be direct on Y or ranges that intersect with Y.

The maximum isolation level, *Serializability*, essentially requires that the transactions operate as if they are completely alone, executing one-by-one in a serial manner. Serializable isolation prevents such phantom data from becoming visible, although the solution varies in implementation and is not as general as the previous isolation levels. (See Section 4.7 for implementations to solve phantom reads.)

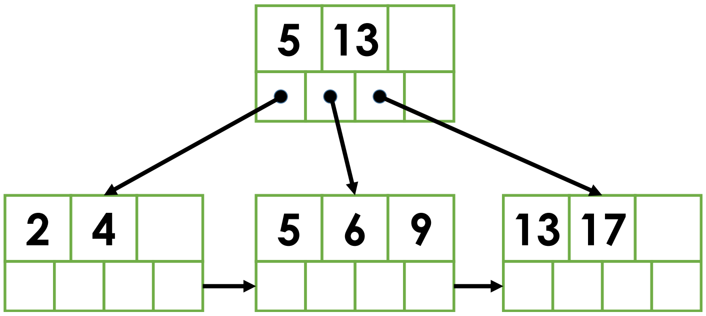


Figure 4.4: Berkeley DB Data Storage (B+ Tree)

4.6 AsR Without Range Queries

Before going into the lower-level details for processing transactional executions, we must first give a brief overview of typical database structures. In abstraction, a database index is generally seen as a hash map, where the keys are IDs of some comparable variable type that is used to retrieve the associated data. In actuality, many implementations of databases utilize a B+ Tree [39, 40] as seen in Figure 4.4. Per each node in the tree, there may be multiple entries and divisions utilizing the object keys in order to allow efficient searching. At the bottom of the tree is a list of all items currently in the structure, sorted in ascending order. The nodes above the list encompass sub-sets of the data.

For transactions to access objects in the tree, they must navigate via the nodes using *latches*, which are short-term synchronization points that are allocated to threads to prevent others from reaching their position. In terms of the database, transactions will latch hand-over-hand, securing the next node while they still hold the current one, which will prevent other transactions from passing and potentially reaching the same data earlier than they do. Once transactions get to the bottom of the tree, they may read the data encompassed by the current slot they are looking at, or may add data to an empty slot in that leaf node's storage. If the node is full, then it must be split and other nodes must be created (and may even be propagated up the tree, changing some of the sorting).

Now, we will proceed with the extensions to the DBMS, beginning with the default Read Committed (RC) level of isolation. Note that this entire section ignores the potential occurrence of phantom reads, which will be addressed in Section 4.7, meaning that Repeatable Read (RR) is the highest isolation level needed. Further, Section 4.6.1 explains the extensions without the possibility of AsR transactions upgrading to stricter levels of isolation. Section 4.6.2 then expands to that capability.

4.6.1 Without Upgrade

Transactions in AsR start at the RC isolation. As a reminder from Section 4.5.1, dirty reads are the anomaly prevented by RC isolation. In order to do so, write-locks must be placed on items when a transaction modifies them, thus preventing other transactions from accessing the items until the lock holder commits or aborts.

But RC isolation, as aforementioned, does not protect against non-repeatable read or lost update anomalies. As explained in Section 4.5.2, the broad solution for these two anomalies is to require transactions to hold read-locks on objects until they commit or abort. But what if those read-locks are not always needed?

Say that two transactions, T_1 and T_2, execute simultaneously: T_1 reads some set of objects while T_2 writes to a sub-set of those same objects. If T_1 does not repeat a read on the written data, nor writes any objects that T_2 has previously read, then both of the transactions can be serialized as $\{T_1 \prec T_2\}$. By the constraints of RR isolation, even though read-locks are a generally simple design, T_2 would be aborted in such a case and cannot complete until T_1 has finished its execution (or has aborted).

In fact, transactions can even deadlock over read-locks. Say that T_1 and T_2 read objects X and Y, respectively. Then their next operations are to write Y and X, respectively. Because of the reversal of the reads and writes, the two transactions depend upon each other due to the read-locks, and both transactions will deadlock if the transactional system allows for blocking on locks. If the system allows for immediate aborts, it is still possible both will see the conflicts at roughly similar times and both will abort. Further, they both may obtain the respective read-locks again, if they are scheduled properly. While this deadlock example may seem overly simple, it is in fact easily possible for applications to have deadlocks in databases, just as it is easy in any general application that utilizes locks.

Taking the above perspectives into account (as well as other cases similar to the example in Figure 4.1), AsR was designed to uphold the correctness of RR isolation without actually holding read-locks for a transaction's lifetime. Further, the solution is relatively simple to implement with an acceptable overhead, and it is general enough that it can be easily adapted to fit different database designs.

As described in Section 4.4, AsR adds meta-data for objects and transactions in the database, but the meta-data's size is independent of the objects or transactions themselves. Because the meta-data is constant and minimal size, there is not an expensive overhead added to the database; and further, the meta-data is not stored inside of indexes with the objects. The only variable that determines the size of meta-data (from the application's beginning) is the maximum amount of concurrent threads allowed in the system.

Concurrent hash maps are attached to each individual database in an application, where the maps connect the primary keys of data objects to individual *readers-writer blocks* (RWBlock). Inside of the RWBlocks, there are two pieces of information: (1) `readers`, an array (sized to the maximum number of concurrent threads) that holds the *transactional contexts* of the latest transactions to read the object in question; and (2) `writer`, the context of the last transaction to write the object. Each space in the `readers` array is allocated to a given thread, and each transaction in the system is assigned a *thread ID*, `tid`, at its initialization. Thus, transactions only edit their thread's position in the array.

Each transaction is given its own context, which contains an array, `invWriters`, used to check whether the given transaction has been invalidated or not. This array is also dependent upon the maximum number of threads, and transactions similarly edit only their assigned positions. When transactions prepare to commit, they observe their `invWriters` array. If any of the transactions contained there have already committed, then the current transaction must abort, as they have been invalidated and may have already read incorrect information.

Algorithm 4: Transaction Operations and Processing (Part 1)

```
Procedure: Read
Input: txn, objKey
Output: Possible Exception
▷ Get meta-data for object and retrieve the last writer
rwb = getOrInsert(objKey, new RWBlock())
rwb.lock()
▷ Check if the writer is active
CheckWriter(rwb.writer)
▷ Make the read visible
rwb.readers[txn.tid] = txn
rwb.unlock()
▷ Continue to normal DB read...

Procedure: CheckWriter
Input: writer
Output: Possible Exception
▷ If the latest writer is active, throw an exception for a conflict
if writer $\neq$ null and writer.active then
    throw ConsistencyException
end if
```

To better understand the modified execution of transactions, Algorithms 4 and 5 display the general read and write operations, along with the necessary meta-data processing.

We begin by observing the `Read` operation. The variables required from the application are the transaction performing the operation, `txn`, as well as the key representing the object, `objKey`. Firstly, the transaction must retrieve the meta-data (i.e., the RWBlock) associated with the object, or it must create a new one if the object does not exist (line 4.3). In order to properly process the RWBlock, `rwb`, transactions must lock it for mutual exclusion (line 4.4).

Next, the reading transaction observes the latest transaction to write the object, and it checks (via the transaction context) if the writer is still active (lines 4.6, 4.11-15). If the writer is active, a *consistency exception* is thrown; otherwise, the transaction continues on. A consistency exception causes the transaction to abort, simply noting that it is unable to continue while the data it needs is not yet updated. To put it differently, it is likely that a transaction will no longer have consistent data if it reads something that was freshly-written during its own execution. Note that this conflict detection prevents the transaction from entering the database index, thus alleviating some of the contention on the database itself.

If no conflict occurs, then the reading transaction marks itself in the `readers` array of the RWBlock, with the position determined by the transaction's thread, `tid` (line 4.8). Lastly, the transaction unlocks the meta-data (line 4.9) and continues with the normal read operation performed by the DBMS.

Now, we move on to the `Write` operation, which requires the same variables as `Read`, with the additional `value` to be written to the object in question. Note that the first conditional piece of code (lines 5.2-11) will not be utilized here, but rather will be expanded upon in Section 4.7.

Similarly to the `Read` function, the transaction must obtain (or create) the RWBlock associated with the object, lock it for mutual exclusion, and perform the conflict check to see if the object has an active writer (lines 5.13-16). However, the new step introduced is the `InvalidateReaders` function (line 5.17). In it, the writing transaction observes all active transactions that have read the current object (lines 5.23-24). As explained earlier, all transactions own an `invWriters` array (shorthanded to `inv` in the algorithm) which holds markers for other transactions that may have invalidated the owner.

Using its thread ID, `tid`, the writing transaction looks at its assigned position within the array (per each reader transaction). If the marked position is empty, or the last writer on the thread did not commit (meaning that it did not actually invalidate the reader), then the current writer will place itself there (lines 5.25-27). Lastly, the transaction marks itself as the latest writer of the object (line 5.19) and unlocks the meta-data (line 5.20), continuing on with the DBMS's normal write operation.

Algorithm 5: Transaction Operations and Processing (Part 2)

```
Procedure: Write
Input: txn, objKey, value
Output: Possible Exception

if new object is being inserted then
    ▷ Must retrieve the immediately next object first
    nextObj = DB.getNext(objKey)
    rwb = getOrInsert(nextObj.key, new RWBlock())
    rwb.lock()
    CheckWriter(rwb.writer)
    InvalidateReaders(rwb.readers, txn)
    rwb.writer = txn
    rwb.unlock()
end if
▷ Check the actual object
rwb = getOrInsert(objKey, new RWBlock())
rwb.lock()
▷ Writer should be null for new objects
CheckWriter(rwb.writer)
InvalidateReaders(rwb.readers, txn)
▷ Mark self as the last writer
rwb.writer = txn
rwb.unlock()
▷ Continue to the normal DB write...

Procedure: InvalidateReaders
Input: readers, txn
Output: Possible Exception

foreach transaction R in readers do
    if R $\neq$ null and R.active then
        if R.inv[txn.tid] is null or did not commit then
            R.inv[txn.tid] = txn
        end if
    end if
end for
```

4.6.2 With Upgrade

Now, we will move into the possibility of AsR transactions upgrading to stricter isolation levels. As previously mentioned, RR isolation is the strictest level required for the DBMS if phantom reads are not possible. In AsR, if a transaction fails to commit in RC isolation for some set number of times, it may need to actually grab a read-lock and exclude other transactions from accessing the data it needs. Certain levels of contention (or transaction schedules) can cause cyclical conflicts that can only be broken by obtaining locks to exclude the other transactions from progressing until some of the work is complete. In such a case, AsR allows the transaction to restart with new isolation settings fit to RR.

But, due to this adapting isolation, it is possible to have RC and RR transactions executing at the same time in the system. Despite the fact that the RR transactions could run without any extra processing, we still must force them to utilize the meta-data in order to keep all transactions correct. If the RR transactions did not invalidate readers when they write objects, then it would be possible for the RC transactions to execute without catching the conflicts.

On the positive side, the current meta-data processing already handles such problems. Any RR transaction that reads an object can still perform the pre-emptive `CheckWriter` function, but it does not have to mark itself in the `readers` array afterwards. Because it will actually take the read-lock on the object, the transaction will already prevent new writers from accessing the object. And when it comes to writing the object, RR transactions must simply mark the current readers as (potentially) invalidated as usual.

With the above processing, AsR is now able to handle all transactions without the possibility of phantom read anomalies. Next, Section 4.7 will extend processing to handle those anomalies.

4.7 AsR With Range Queries

Phantom reads, as described in Section 4.5, are when a transaction performs multiple read operations during its lifetime, and a recently-created object does not appear in all of the read operations that should have covered it. Hence, the reading transaction cannot be considered atomic if it occurs both before and after the operation that created the object in question.

Serializability (SR) is the highest isolation level considered for the DBMS, and it prevents the occurrence of these phantom reads. The actual implementation of SR isolation can vary between systems and is not as direct as the RC and RR isolation levels. In Berkeley DB, SR isolation utilizes *next-key locking.* As described at the beginning of Section 4.6, the data in a B+ tree is stored in a sequential list at the bottom of the tree. Next-key locking is a protocol that requires both the current object being read or written, as well as the item immediately next to it, to both be locked during an operation. More specifically, the object being operated upon, as well as the object it points next to in the list, will be locked.

This protocol solves the problem of phantom reads, as when a transaction is inserting a new item into a database, it must first determine the item's location in the data list. When doing so, the transaction must first lock the object that will be in front of the inserted item. If a transaction has already read-locked the next object, then the new item cannot be inserted, as a preventative measure.

In order to account for the next-key locking without actually utilizing locks, additional processing is added to the `Write` function as seen in Algorithm 5. The previously-skipped piece of code (lines 5.2-11) is now used to solve this anomaly. If a new item is to be inserted into the database, the transaction must actually go into the DBMS to determine what the immediately next object will be (lines 5.3-4). The rest of the block is identical to the normal processing, as the writer updates the meta-data for the next object and invalidates the readers (lines 5.5-10). Afterwards, the transaction performs the rest of the `Write` function on the object being written (although no meta-data should need to be processed, as it should not exist until now), inserting the object into the database.

4.8 Evaluation

Berkeley DB Java [26, 40] serves as the basis for AsR's prototype implementation, and the testbed is a 64-core machine which has 4 AMD Opteron 6272 processors, each with 16 cores running at 2.1 GHz, 16MB of L2 cache, and 128 GB of RAM. Machines of this type were accessed through PRObE [41], a public infrastructure.

Evaluation of AsR utilized three commonly-used applications: TPC-C [16] and TPC-W [30], because they represent complex workloads, and Bank, because it is a lightweight and customizable synthetic benchmark. In order to present more insightful and fair results, and to select challenging configurations, the evaluation disabled (Section 4.8.1) and enabled (Section 4.8.2) the possibility of having the phantom read anomaly. The reasoning for this decision is because phantom reads are tightly related to the presence of range queries and insert operations performed by the application. With phantom reads disabled, the strongest and sufficient isolation level provided is Repeatable Read; otherwise, it is clearly Serializability. Hence, the default Read Committed (RC) and Repeatable Read (RR) isolation levels of Berkeley DB were tested in every benchmark and setting, while Serializability (SR) was only tested with phantom reads enabled. Note that unedited Read Committed isolation is simply used as a rough boundary for performance, as its actual execution cannot be consistent for any of the benchmarks. Similarly, Repeatable Read is consistent when phantom reads are disabled, but it is not consistent when they are enabled.

Benchmarks are notably configured to generate medium to high contention. With little to no contention, the default locking performed by transactions would cause few conflicts, thus the benefits of AsR would be reduced. Similarly, three different settings for AsR were tested. One is the default setting that does not allow the transactions to upgrade to stricter isolation levels, while the other two settings allow for upgrades, but at differing abort thresholds (typically set at 3 and 5 in these experiments).

Lastly, before the evaluation details begin, there were some trends discovered across the experiments that can be generalized here. The Berkeley DB system did not show great scalability even in non-contentious cases. In many of the results, the throughput of the system increased as a small amount of threads was added, but the system quickly bounded due to some limited internal resources, resulting in additional overhead. As the number of threads using the database increases, the time to read or write an item extends, even if there are enough items in the database such that the overlap in usage between the threads is very low. Additionally, AsR's overhead (on top of the database's internal overhead) sometimes limits AsR for a lesser amount of threads, but still proves its benefits of outlasting the other isolation levels.

In this next section, we begin our evaluation on all of the three benchmarks mentioned above, but they are configured not to use the range query database primitives.

4.8.1 Without Range Queries

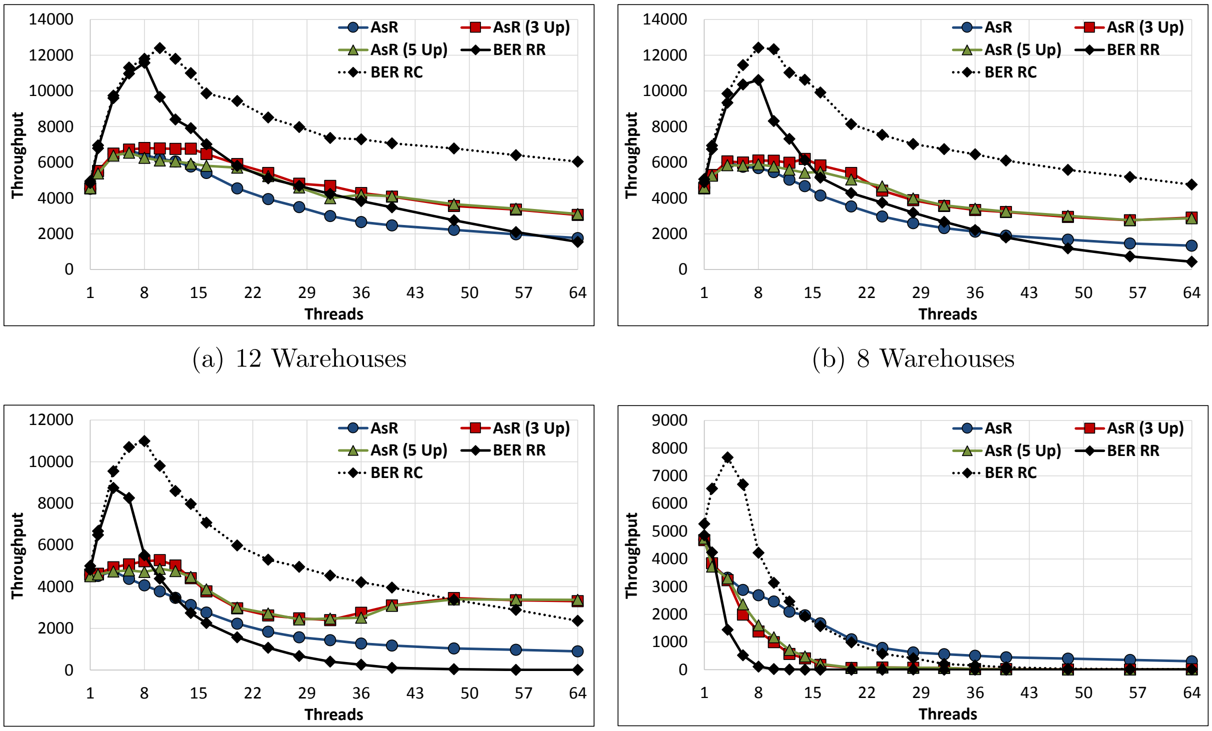
TPC-C

TPC-C [16] is a larger benchmark simulating stock warehouses with item orders and deliveries. The benchmark initializes by creating a set number of warehouses, allocating districts, customers, items, and other information to each one individually. There are five main transactional operations that can be performed: *i)* placing new warehouse orders; *ii)* checking the status of orders; *iii)* updating system information and customer balances with order payment and confirmation; *iv)* checking the stock levels of warehouses and refilling as necessary; and lastly, *v)* delivering the items required to fulfill batches of orders. Of the five operations, the creation of new orders and the payment processing are the most frequent, and from the operational profile, 92% of the transactions executed are write transactions. Lastly, the main way to alter the contention in the benchmark is by simply decreasing the number of warehouses available while increasing the number of user threads.

Figure 4.5 shows the general throughput of transactions starting with 12 warehouses and going down to just 1 warehouse. Each experiment also starts with 1 user thread and increases up to 64, all upon a centralized database. As can be seen in all of the plots, the throughput for all types of isolation decreases as the number of threads increases, and it goes down at a faster rate the lesser the number of warehouses. The RC transactions roughly hold the best "performance" in most of the plots, but that is not exactly true; as mentioned before, RC transactions are actually incorrect for such an application due to their rules, thus the higher throughput is not realistically achievable in these examples.

In these experiments, AsR is run at its default setting as well as allowed to upgrade to higher isolation levels after 3 or 5 aborts have occurred upon a single transaction (meaning that it is failing regularly at its current setting). AsR performs better than RR isolation generally, but its processing holds it back in less-conflicting scenarios (i.e., low numbers of threads with 12 and 8 warehouses). However, RR transactions drops rather quickly in all of the plots as more concurrent transactions are added, thus showcasing AsR's ability to outlive the default concurrency processing.

One interesting thing to note here is that AsR's upgradeable transactions perform better than the default, fixed-isolation AsR mode. The advantage of being able to change isolation



(a) 12 Warehouses

(b) 8 Warehouses

(c) 4 Warehouses

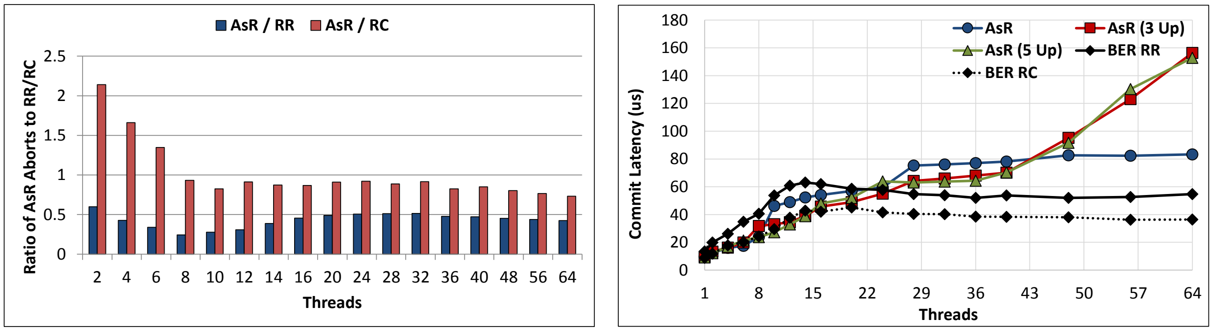
(d) 1 Warehouse

Figure 4.5: Throughput of TPC-C (No Range Queries)

levels varies in magnitude per application (and can even turn into a disadvantage in some), but here the focus is on the main warehouse objects. While all transactions, regardless of the type, must access the warehouses themselves, only some of them must write and update the high-level information of the warehouse. Depending on the current layout of the transactions and their types, as well as the system's contention, transactions looking to explicitly write the warehouses will face more problems, as the number of users will generally far exceed the number of warehouses. Because of this issue, if an AsR transaction upgrades to RR isolation, it is able to hold a read-lock on the warehouse object it needs. Thus, it is able to prevent other transactions from invalidating the information it requires to perform its update.

Next in Figure 4.6 are two additional statistics for TPC-C running with 12 warehouses. The first (Figure 4.6(a)) is a bar graph showing the aborts AsR faces in proportion to the default RC and RR isolation levels. Explicitly, the value per bar is the ratio of the number of aborts AsR encountered vs. RC or RR at each number of threads. While the ratio between AsR and RC is higher than with RR, it is again not a direct comparison like the one against RR. All of the results relative to RC are meant as general upper boundaries in performance.

As can be seen in the plot, the higher the contention, the lower the abort ratio actually is. In comparison to RR, this smaller ratio makes sense, as RR is stricter and should incur more



(a) Abort Breakdown Against RC and RR

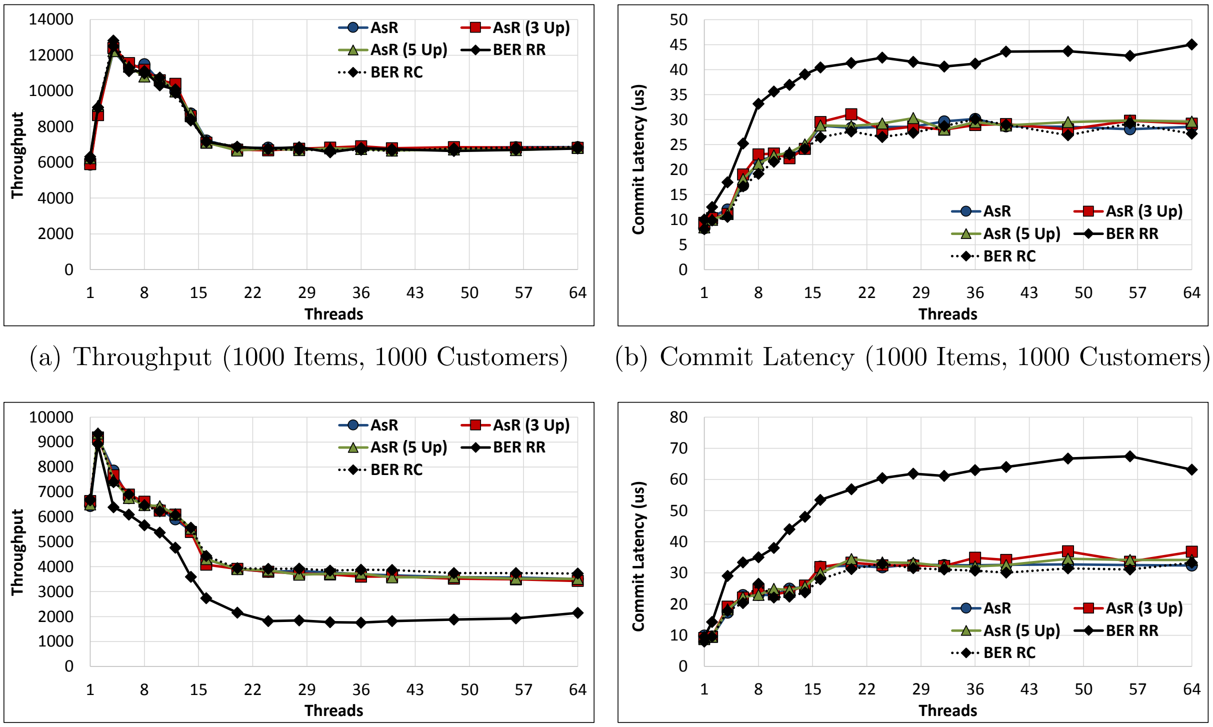
(b) Commit Latency

Figure 4.6: Additional Statistics for TPC-C with 12 Warehouses (No Range Queries)

aborts than AsR or RC should. The ratio decreases to roughly 1/4 of the amount of aborts at 8 threads, but then levels back out to about 1/2. That drop occurs because RR's conflicts increase at a faster rate up to a certain point, then slow down because of two main reasons: *i)* the higher the contention, the longer it takes the database to complete read and write requests (thereby making it take longer for a transaction to potentially abort); and *ii)* in the centralized system, the actual communication takes longer once the work must go outside of a single socket. Then as the contention increases and levels out the system's performance, the number of aborts increases for both RR and AsR at a slower, similar rate.

Against RC isolation, however, AsR still has less aborts. The ratio is close to 1 (and the lowest it goes to is 3/4), which is expected as AsR is a slightly heavier form of RC isolation. While AsR having less aborts may seem odd, it is mostly due to the earlier conflict detection. Transactions can abort earlier than they would in RC isolation, and can potentially restart and complete their work faster. Depending on the flow of execution, transactions restarting earlier could allow them to get through their work without encountering another conflict. Further, transactions must synchronize on the meta-data but they do not immediately abort if what they need is currently locked. The reasoning is that the meta-data is simply locked for the two main operations: if it is locked for a read, then the waiting transaction has no conflict; if it is locked for a write, then the waiting transaction has no conflict if the writer aborts, and could still have no conflict if it reads the writer's information and does not detect a problem.

Lastly, Figure 4.6(b) shows the commit latency for the various modes as the number of threads increases. As can be seen, the default isolation levels have the lowest commit latencies; but note, these times are only measured on actual commit operations and not for failed transactions. AsR's normal version has a higher latency because of the synchronization required to perform validation at commit time (while general database transactions must perform operations like release locks and clear internal data). But the upgradeable AsR has the highest latency because there is extra overhead added for validation for some transactions while others are at RR (or even SR) isolation levels. Despite the higher commit latency, the adaptable transactions still held better throughput because they are able to get around the contention (i.e., some transactions can succeed at the lowest level of isolation while others need to be stricter to get through). Essentially, there is a rough balance struck on what isolation levels the transactions must use.



(a) Throughput (1000 Items, 1000 Customers) (b) Commit Latency (1000 Items, 1000 Customers)

(c) Throughput (10 Items, 1000 Customers) (d) Commit Latency (10 Items, 1000 Customers)

Figure 4.7: Throughput and Commit Latency of TPC-W (No Range Queries)

TPC-W

TPC-W [30] is a benchmark that emulates a commerce website, and can be thought of similarly to TPC-C but in a broader form. There are 14 different types of transactions to be executed that function as a user or administrator navigating the website, looking at and purchasing objects, performing maintenance, etc. In terms of data, there are 8 different types of objects representing *customers*, *items*, *orders*, and so on. Transactions are completed one-by-one and leave a given *state* in the system in order to influence the next operations performed. For instance, if a customer just observed some search results, that state influences the chances to add the item(s) to their cart or to return to the query page or home page.

The main changing settings that influence the system's contention, besides the number of threads (i.e., *active* users and administrators), are the number of items available as well as the number of customers who have accounts with the website. One thread, until some endpoint is reached, is meant to simulate one customer or one administrator performing their necessary operations.

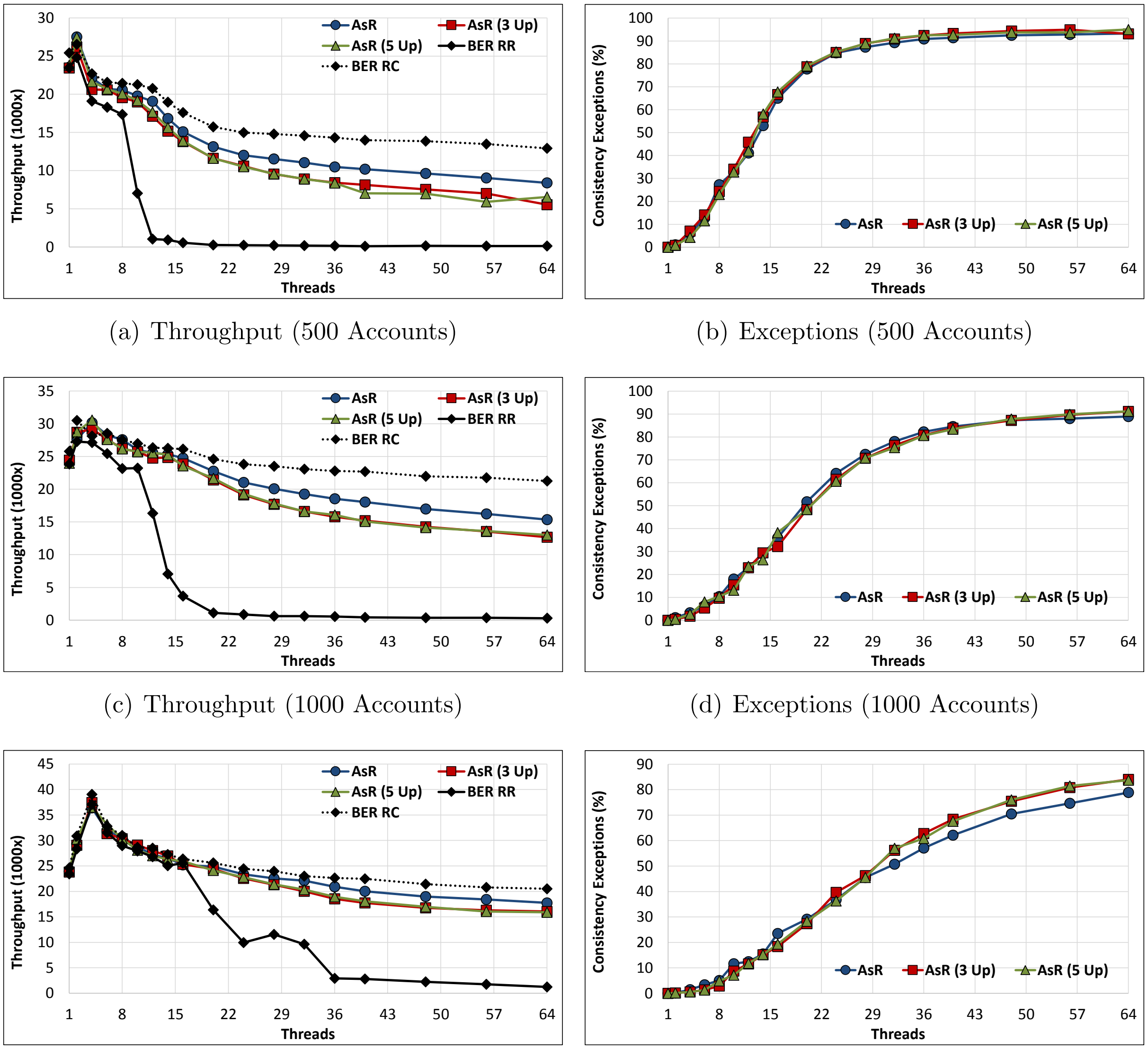
Two experiments are shown in Figure 4.7, displaying how RR isolation can allow for more transactions to abort than may be necessary. The top row has the throughput and commit latency with 1000 items and 1000 customers, while the bottom row holds the same information but for 10 items and 1000 customers. Note, as said above, 1000 customers only means that there are 1000 customer accounts, not 1000 customers using the system at once.

With so many customer accounts in comparison to the number of active users, as well as the large number of items available, all of the isolation levels actually hold the same performance in Figure 4.7(a). In this setting, TPC-W has very low contention and is meant to demonstrate that AsR does not fall below the other isolation levels even if it cannot outperform them. Even though RR's commit latency is slightly higher in Figure 4.7(b), it balances out and does not affect its throughput that drastically. In terms of the actual throughput values, RR is approximately 200 lower in throughput than the others, which comes out to about a 3% difference.

When we bring the number of items down to 10, it decreases the throughputs of all isolation levels, but actually causes RR to drop relative to the other modes as well (Figure 4.7(c)). Despite such a low number of items, the performance of the system does not decay as much as with TPC-C, for instance. The reasoning here is that the actions are all determined by a non-uniform random function and depend upon the current states of the threads, as mentioned earlier. Not every transaction is actually performing a conflicting action; and in fact, it is possible that there are instants in time where the majority of the transactions may not be performing writes, depending on their states. From that, we can also see that RR performs worse because of its strictness with read-locks. Many transactions will likely be overlapping on some of the same data, thus preventing proper writes because of read-locks. The read-locks are not necessary here as the item data is refreshed with new transactions (e.g., like when a user changes pages to look at other search results or a specific item).

Bank

The Bank benchmark is the simplest tested, and has only two main operations: *balance check*, which opens a set of bank accounts and observes their values; and *transfer*, which withdraws money from one account and deposits it into another. In the testing performed, a write-heavy workload was used to show the general difference between the meta-data processing and traditional synchronization. As detailed in the introduction of Chapter 4, low- or no-conflict workloads will gain smaller benefits from AsR as the general synchronization would also not conflict as much. The number of bank accounts available mainly affects the contention in the system, as does the number of user threads.



(a) Throughput (500 Accounts)

(b) Exceptions (500 Accounts)

(c) Throughput (1000 Accounts)

(d) Exceptions (1000 Accounts)

(e) Throughput (2500 Accounts)

(f) Exceptions (2500 Accounts)

Figure 4.8: Throughput and Conflict Exceptions of Bank (No Range Queries)

As shown in Figure 4.8, there are three pairings of plots. Each row represents a specific number of bank accounts allocated to the system (500, 1000, and 2500). The first column showcases the throughput of AsR as well as RC and RR isolation transactions (shown in a scale of thousands). The second column displays, for AsR, the percentage of transactions that threw consistency exceptions. In other words, it is the portion of transactions that caught conflicts through AsR's meta-data processing as opposed to the database's internal processing. The x-axis in the plots is simply the number of user threads active.

As can be seen in the left column, RR's throughput drops rather quickly as opposed to RC and AsR, but holds out for a longer period of time if there are more bank accounts (e.g., it drops drastically around 8 threads in Figure 4.8(a) but drops at a slightly slower rate around 16 threads in Figure 4.8(e)). In this benchmark, RC seems to holds the best throughput, and as the number of accounts rises, AsR gets closer to that boundary line. Here, the upgradeable version of AsR is not the best (as with TPC-C), although it is not much worse. In fact, as more accounts are available, the gap closes between the different versions of AsR as well.

In the right column, we can see that higher contention results in many more of the transactions throwing consistency exceptions. At 64 threads in all of the plots, roughly 95%, 90%, and 80% of the transactions throw exceptions for 500, 1000, and 2500 bank accounts, respectively. With lesser accounts, the faster the exception rate climbs. For example, at 8 user threads, the percentage is roughly 33% in Figure 4.8(b), but it is about 5% in Figure 4.8(f). What these trends show is that AsR detects more conflicts through its processing than through the database in high-contention scenarios. Because of this capability, it helps AsR outlive the transactions that rely solely on the database's internal processing.

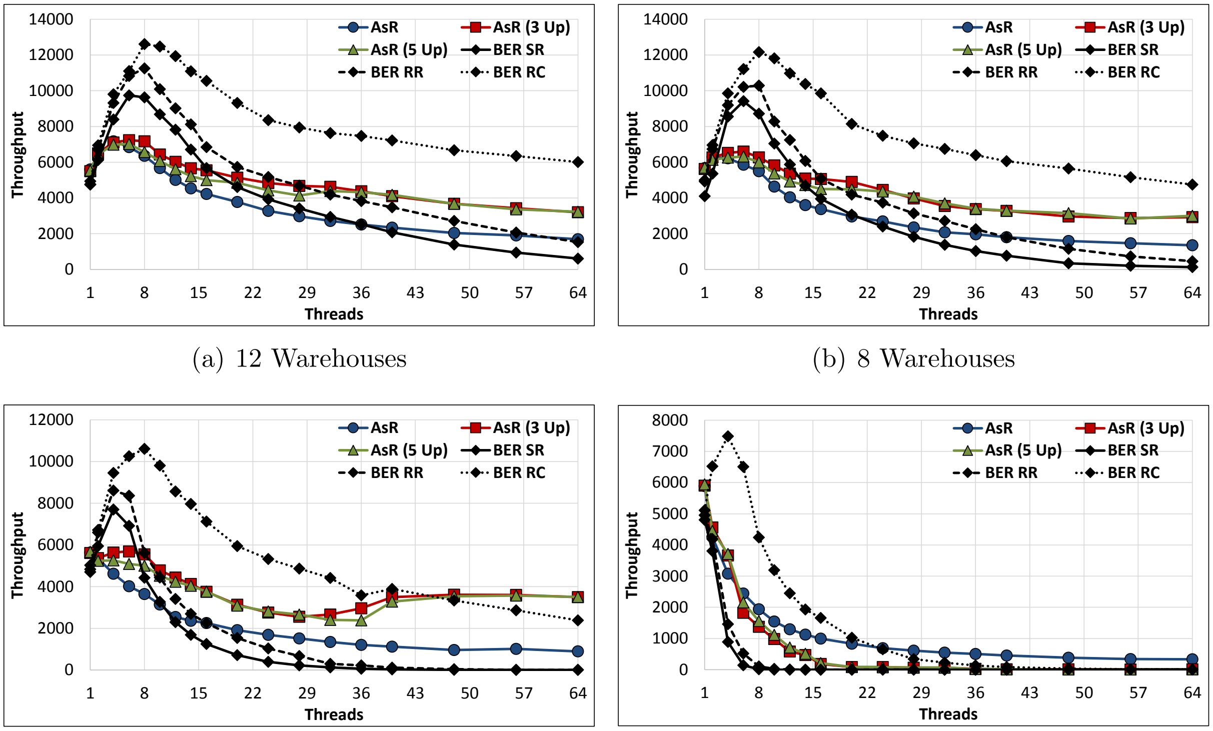
4.8.2 With Range Queries

Next, we revisit all of the benchmarks with the capability of range queries enabled, producing different application executions. As a result, the system performance is altered for all isolation levels. Further, RC and RR isolation are now invalid, thus Serializability (SR) is included for correctness. However, those two lower isolation levels were still utilized in the experiments as boundaries, similar to how RC was used without range queries.

TPC-C

There are mainly two range queries performed by TPC-C, done during order status transactions and stock check transactions. The former will simply look across a set number of current customer's orders, observing all of their statuses. The latter observes a number of most-recent orders, looking at all of the items requested in the orders, checking the stocks of each one. If the stocks of any of the items are below a limit, then they are set to be refilled. In both of these range queries, only a single customer or order ID is required to search. Without a database range query primitive, specific IDs can be given to individually request customers or orders, but not every ID may exist. This method was used for the results in Section 4.8.1. With such a primitive, the search will only return valid information (e.g., the search can explicitly find the last 20 *existing* orders).

Figure 4.9 mirrors the throughput results shown in Section 4.8.1 (in Figure 4.5) for TPC-C, but now with these new range-based operations. As can be seen, AsR still outlives the correct isolation level (here, SR) as the number of threads increases in the system. In fact, it is still able to outlive RR isolation with these range queries, even though RR does not



(a) 12 Warehouses

(b) 8 Warehouses

(c) 4 Warehouses

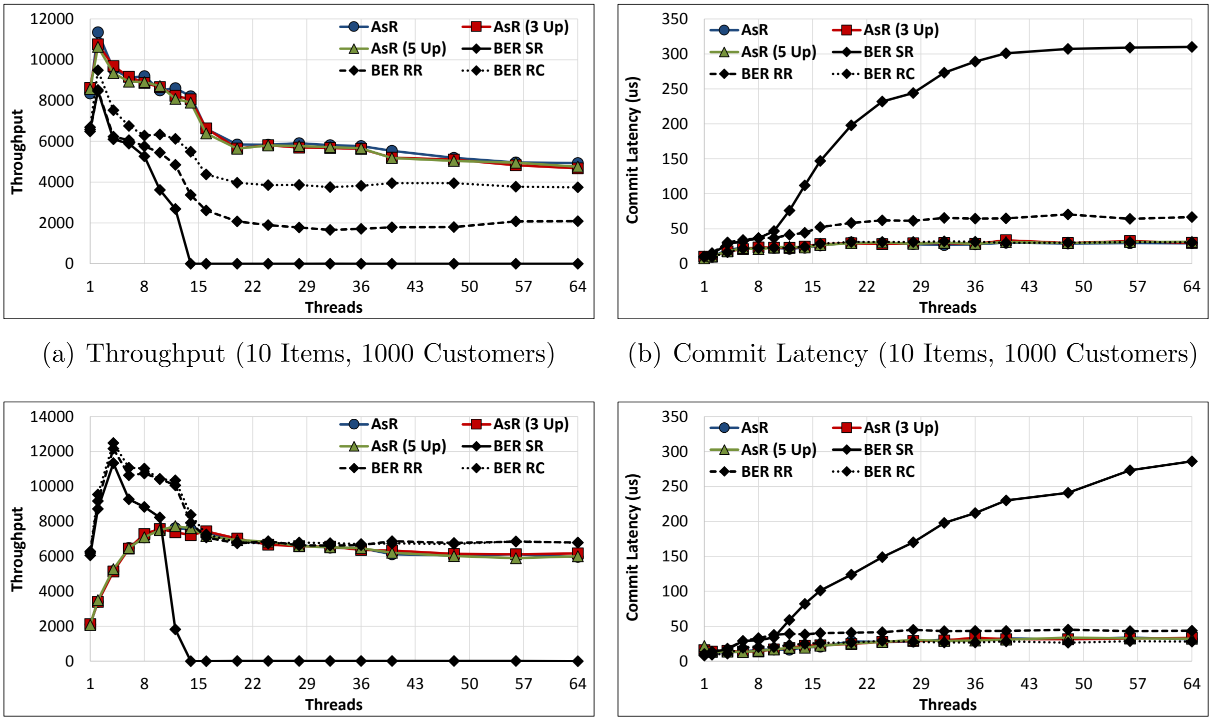
(d) 1 Warehouse

Figure 4.9: Throughput of TPC-C (Range Queries)

add the extra processing employed by SR. The weakest isolation level utilized, RC, generally holds the best results except in really contentious scenarios (like with 1 warehouse, or greater than 32 threads with 4 warehouses). However, as said earlier, its results (as well as RR's) are incorrect in execution, hence its highest throughput is not properly achievable.

As with the non-range experiments, the upgradeable version of AsR performs better because of the warehouse contention points, enabling some of the transactions to pass at an adapted version of RC while others can apply RR as needed. In comparison to Figure 4.5, the results are not drastically different as the range queries are not a giant part of the TPC-C benchmark. In fact, many of the isolation settings follow very similar trends, although the throughputs are generally lower than they were without range queries (e.g., at 4 warehouses and 8 threads, RC is roughly 11,000 without range queries and 10,500 with range queries).

The most important trend to notice is that SR, the only default isolation level capable of handling the range queries, is the lowest in performance, because the contention is so high that SR drops to a point where almost no transactions can get through.



(a) Throughput (10 Items, 1000 Customers)

(b) Commit Latency (10 Items, 1000 Customers)

(c) Throughput (1000 Items, 1000 Customers)

(d) Commit Latency (1000 Items, 1000 Customers)

Figure 4.10: Throughput and Commit Latency of TPC-W (Range Queries)

TPC-W

For TPC-W, the range queries added are mostly user searches, e.g., looking at best selling items or returning sets of items by author, subject, or title, or they are transactions used by administrators to observe multiple orders. Figure 4.10 shows the same settings as in Figure 4.7, but in the opposite order (i.e., 10 items is shown first, then 1000 items). Here we can see the current limitations of the range query processing within AsR, as the transactions are required to go into the actual database to emulate next-key locking through the meta-data (as detailed in Section 4.7).

In Figures 4.10(a) and 4.10(b), we see the throughput and commit latency of the various isolation settings where the TPC-W benchmark is allocated with 10 items and 1000 customer objects. In order to demonstrate a similar case as with Figures 4.7(c) and 4.7(d), AsR is able to hold much better throughput than the default isolation settings, even above RC as it can abort early if necessary due to meta-data conflicts, thus allowing for faster recovery. Further, it highly outperforms SR as the strict isolation mode fails to let many transactions pass once there are 16 or more threads. SR holds the largest commit latency as well, which drastically increases with the number of threads. Note that the introduction of range queries does not cause that drastic of a change, as the searches are over a small amount of items, and as most (not all) are read-only, they only serve to lower the performance of RR and SR with their read-locks, due to the heavy overlap between many transactions.

With 1000 items (Figures 4.10(c) and 4.10(d)), the results are slightly more interesting. While the number of available items is 2 orders of magnitude more, SR still performs very poorly (and in fact drops throughput at a faster rate) while AsR actually gets off to a worse start. The reasoning here is that the range queries performed are rather large (50 to 100, on average); thus, with 10 items available before, they were simply limited to 10 actual items. Because of this difference, the internal processing of these queries is now heavier, requiring more work and time. Hence, SR's processing encounters even more conflicts and takes longer to redo transactions, damaging its performance more than before.

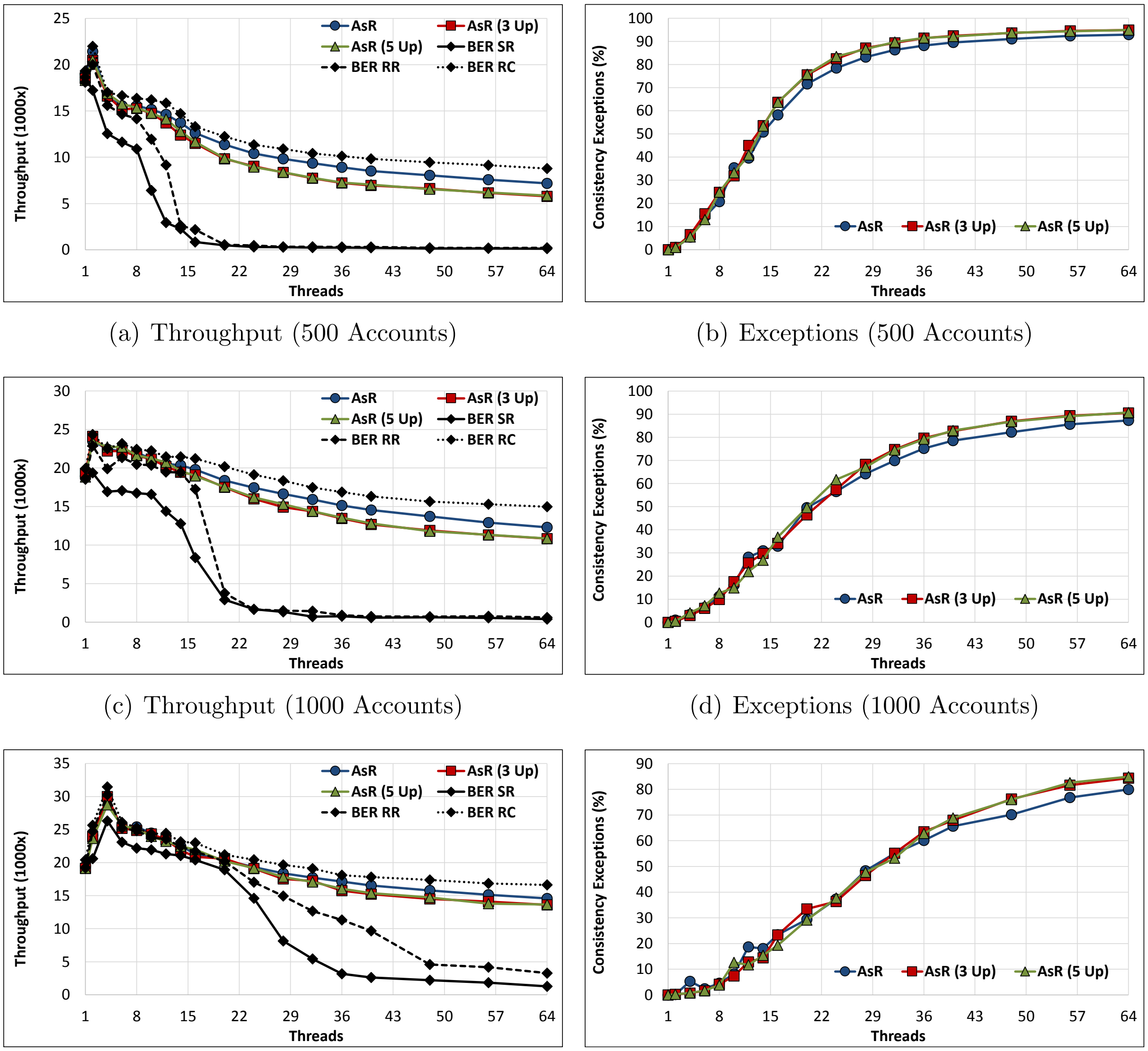
AsR faces a similar issue. Due to its extra processing required for the emulated next-key locking with meta-data, much more time is required to perform range queries as the ranges themselves increase. With only 2 threads in the system, AsR's throughput is roughly 1/3 of what it was with 10 items in the system. Despite this problem, AsR is able to improve again as more threads are added—more parallel work overcomes the length of time required for certain transactions. And it does not take many threads for AsR to catch up to RC and RR in performance (about 14 or 16); and in fact, its throughput is a few hundred better than in Figure 4.10(a) because there is less contention. Similarly, RC and RR improve their performance with lesser contention, and both slightly exceed AsR at a higher number of threads. But again, they do not properly handle range queries to prevent phantom reads.

Bank

Modifications were made to the Bank benchmark in order to allow for range queries. A given chance parameter was added to encourage multi-account transfers to occur. Transfers are only between two accounts at one time, and singular transfers are typically considered one transaction. But by using range queries to collect multiple accounts, transfers were extended to take more than one pair of accounts, thereby performing multiple transfers in one atomic transaction. These larger transactions ran with 20% chance in the experiments shown below.

Figure 4.11 is comparable to Figure 4.8, utilizing 500, 1000, and 2500 account objects in the system. The throughput is shown in the left column for all isolation levels, while the consistency exceptions thrown by AsR are shown in the right. Nearly all of the trends follow similar patterns as before, although the throughputs are decently lower than without range queries (e.g., with 2500 accounts, the maximum throughput is nearly 40,000 without range queries and just over 30,000 with range queries).

The difference between RC and AsR is lower than before. For instance, with 500 accounts, AsR is 81% of the RC's throughput at 64 threads. Without range queries, AsR is 65% of RC's throughput at 64 threads. On the other hand, RR and SR both follow the similar



(a) Throughput (500 Accounts)

(b) Exceptions (500 Accounts)

(c) Throughput (1000 Accounts)

(d) Exceptions (1000 Accounts)

(e) Throughput (2500 Accounts)

(f) Exceptions (2500 Accounts)

Figure 4.11: Throughput and Conflict Exceptions of Bank (Range Queries)

trends of dropping to bare throughput rather quickly, although they pick up as more accounts are added to the system. AsR also follows the same contention trends with its consistency exceptions—the more accounts, the slower the percentage of exceptions climbs. Note that the maximum values at 64 threads are nearly the same as before: roughly 95%, 90%, and 85% of the transactions were throwing these exceptions.

Chapter 5

Conclusions and Future Work

5.1 Thesis Summary

In this thesis, we presented two main research contributions. The first is SPCN, an algorithm that can parallelize internal nested transactions while supplying two types of automatic execution and conflict processing (Strict and Relaxed). To allow the parallelization without requiring developers to modify their applications, the nested transactions are enforced in a serialized order (the order in which they are defined in the application). This order ensures that later nested transactions will read any new information created by their previous nested transactions.

If any data is not up-to-date, then the parallelization has allowed a conflict, which SPCN will process on its own and resolve. Strict SPCN enforces the completion of operations sequentially, and will stall future nested transactions from committing their execution until all previous nested transactions have committed. Doing so can sometimes prevent work from being completed as soon as possible, but allows for lighter conflict processing as single instances of each data object are shared between all nested transactions. Relaxed SPCN, on the other hand, allows nested transactions to distribute their new data immediately, without waiting on previous nested transactions. However, this protocol can result in conflicts being detected after a nested transaction has been committed, because it did not have access to all operations executed at its time of validation. To detect these conflicts, multiple versions of objects must be held to ensure that invalid writes are not performed on single instances of data. Nested transactions can perform validation on other committed nested transactions to remove information that is no longer valid and enforce them to restart for correctness.

Both versions of SPCN allow it to overcome the limitations of traditional (sequential) nesting, and while the additional meta-data processing is required to maintain correctness, SPCN still significantly improves an application's performance. In the four benchmarks utilized, SPCN outperforms sequential closed nesting by 1.78× up to 3.78× in throughput.

The second contribution presented is AsR, As-Serializable transactions, an adaptable processing system which can be added over any existing database or transactional system to provide the isolation of Serializable transactions through relaxed rules. By trading internal system processing for simpler, external meta-data processing, AsR can overcome the performance drawbacks of Serializability and allow more transactional executions to successfully complete. AsR keeps track of the dependencies between nested transactions, creating an abstract dependency graph. The graph is not fully-connected and does not track every type of dependency, thereby making the algorithm slightly pessimistic. However, manipulating full graphs for processing is very difficult to perform efficiently and generally creates more overhead for applications, lowering the algorithm's benefits.

In particular, AsR tracks read-write anti-dependencies. When a transaction writes objects that have been read by other currently-active transactions, the writer will mark the readers' meta-data, declaring anti-dependencies. If the writer transaction completes its operations and commits before any of the reader transactions, then the readers have executed with invalid information. Depending on their operations, the readers can still be synchronized before the writer to fulfill Serializability; but without tracking more extensive dependency chains, it is not possible to guarantee their correctness. Further, the longer the reader transactions are, the more unlikely it is that they can commit correctly. Thus, AsR enforces these transactions to abort and restart their work.

As seen through the evaluation of three benchmarks, AsR can handle high-contention executions significantly better than typical synchronization, outlasting traditional isolation levels as their performance drops due to their inability to allow sufficient progress for transactions.

5.2 Conclusions

The significance of these two works lies in their performance capabilities, their adaptability, and their simplicity. Both share a large goal of high programmability, so minimal work is required by developers to apply them to their own systems, and no changes to an application should generally have to be executed. Because of these characteristics, SPCN and AsR are exceptionally useful, easily granting large performance increases.

Further, both algorithms can be widely-used in a variety of systems. In the implementations shown here in the thesis, SPCN utilized a distributed transactional memory system, while AsR utilized a centralized database management system. However, both of these contributions are large in scope and can adapt to different systems to provide similar improvements.

Therefore, SPCN and AsR are largely efficient and accessible algorithms that continue the important trends of parallelism with today's multi-core architectures.

5.3 Future Work

There are a number of extensions applicable to the solutions illustrated in this thesis, providing suggestions for future work. Firstly, SPCN is only implemented into a general, distributed network. Because of the algorithm's usability, SPCN can be extended to fit fault-tolerant, distributed concurrency control systems. In terms of performance improvements, the Relaxed form of SPCN can be upgraded for faster processing. As Relaxed checks for conflicts and executes the commit methods of sub-transactions, wait-free data structures can be applied to the algorithm in order to increase the speed of execution. By handling these execution drawbacks, Relaxed may see better performance than Strict in more of the experiments run.

The second contribution, AsR, can be implemented into different systems to provide a better understanding of its scalability, as the given database system and experiments utilized do not allow any of the isolation levels to truly increase performance. Although the experiments shown here utilize a centralized system, AsR can be deployed in distributed systems as well. Further, AsR can be adapted to support more recent and growing isolation levels, such as Snapshot Isolation, allowing it to handle even more types of applications. Thus, AsR can form better compliance with multi-versioned concurrency controls, which are widely included in recent DBMSs.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114–117, Apr. 1965.

[2] D. P. Rodgers, "Improvements in multiprocessor system design," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, (Los Alamitos, CA, USA), pp. 225–231, IEEE Computer Society Press, 1985.

[3] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Comput. Surv.*, vol. 15, pp. 287–317, Dec. 1983.

[4] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 125–143, Mar. 1977.

[5] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 455–495, July 1982.

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, (New York, NY, USA), pp. 289–300, ACM, 1993.

[7] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, (New York, NY, USA), pp. 204–213, ACM, 1995.

[8] J. E. B. Moss, "Nested transactions: An approach to reliable distributed computing," tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[9] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and architecture sketches," *Sci. Comput. Program.*, vol. 63, pp. 186–201, Dec. 2006.

[10] A. Dhoke, B. Ravindran, and B. Zhang, "On closed nesting and checkpointing in fault-tolerant distributed transactional memory," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, (Washington, DC, USA), pp. 41–52, IEEE Computer Society, 2013.

[11] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Implementing and evaluating nested parallel transactions in software transactional memory," in *SPAA*, pp. 253–262, 2010.

[12] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *PPoPP*, pp. 163–174, 2008.

[13] N. Diegues and J. Cachopo, "Practical parallel nesting for software transactional memory," in *DISC*, pp. 149–163, 2013.

[14] A. Dhoke, R. Palmieri, and B. Ravindran, "An automated framework for decomposing memory transactions to exploit partial rollback," in *2015 IEEE 29th International Parallel and Distributed Processing Symposium, Hyderabad, INDIA, May 25-29, 2015*, 2015.

[15] A. Turcu, B. Ravindran, and R. Palmieri, "Hyflow2: A high performance distributed transactional memory framework in scala," in *PPPJ*, pp. 79–88, 2013.

[16] TPC Council, "TPC-C Benchmark," February 2010.

[17] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: A benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, 2007.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, pp. 143–154, 2010.

[19] Amazon Inc. Elastic Compute Cloud, December 2014.

[20] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Middleware*, pp. 187–207, 2012.

[21] S. M. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *PPoPP*, pp. 179–188, 2011.

[22] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*, pp. 455–465, 2012.

[23] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[24] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems*, vol. 31, no. 1, pp. 133–160, 2006.

[25] S. Peluso, P. Romano, and F. Quaglia, "SCORe: A scalable one-copy serializable partial replication protocol," in *Middleware*, pp. 456–475, 2012.

[26] M. A. Olson, K. Bostic, and M. Seltzer, “Berkeley DB,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’99, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999.

[27] M. Widenius and D. Axmark, *MySQL Reference Manual.* Sebastopol, CA, USA: O’Reilly & Associates, Inc., 1st ed., 2002.

[28] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’95, (New York, NY, USA), pp. 1–10, ACM, 1995.

[29] C. H. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, vol. 26, pp. 631–653, Oct. 1979.

[30] TPC Council, “TPC-W Benchmark,” April 2005.

[31] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The notions of consistency and predicate locks in a database system,” *Commun. ACM*, vol. 19, pp. 624–633, Nov. 1976.

[32] A. Fekete, D. Liarokapis, E. O’Neil, P. O’Neil, and D. Shasha, “Making snapshot isolation serializable,” *ACM Trans. Database Syst.*, vol. 30, pp. 492–528, June 2005.

[33] M. J. Cahill, U. Röhm, and A. D. Fekete, “Serializable isolation for snapshot databases,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’08, (New York, NY, USA), pp. 729–738, ACM, 2008.

[34] S. Revilak, P. O’Neil, and E. O’Neil, “Precisely serializable snapshot isolation (pssi),” in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE ’11, (Washington, DC, USA), pp. 482–493, IEEE Computer Society, 2011.

[35] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues, “Making georeplicated systems fast as possible, consistent when necessary,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, (Berkeley, CA, USA), pp. 265–278, USENIX Association, 2012.

[36] C. Li, J. Leitao, A. Clement, N. Preguica, R. Rodrigues, and V. Vafeiadis, “Automating the choice of consistency levels in replicated systems,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, (Berkeley, CA, USA), pp. 281–292, USENIX Association, 2014.

[37] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, “Consistency-based service level agreements for cloud storage,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP ’13, (New York, NY, USA), pp. 309–324, ACM, 2013.

[38] Team, PostgreSQL Development, *PostgreSQL Programmer's Guide.* iUniverse, Incorporated, 2000.

[39] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, pp. 121–137, June 1979.

[40] Oracle, "Berkeley DB Java Edition Architecture," 2006.

[41] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "PRObE: A thousand-node experimental cluster for computer systems research," vol. 38, June 2013.