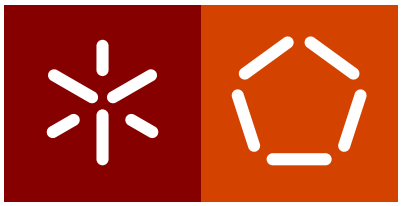
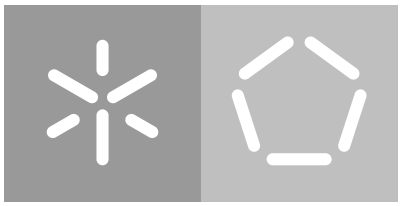


Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Maia Brandão

Convergent types for shared memory

February 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

José Pedro Maia Brandão

Convergent types for shared memory

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Carlos Baquero, University of Minho, Portugal
Annette Bieniusa, University of Kaiserslautern, Germany

February 2019

ACKNOWLEDGEMENTS

I would like to thank my advisors, Carlos Baquero and Annette Bieniusa for their advice and availability. A special thanks to Deepthi Akkoorath for the co-working on the Global-Local View publication.

Last but not least, I would like to thank Vitor Enes Duarte for all the insightful discussions and André Duarte from Paddy Power Betfair for providing me the conditions to finish this document while working.

ABSTRACT

It is well-known that consistency in shared memory concurrent programming comes with the price of degrading performance and scalability. Some of the existing solutions to this problem end up with high-level complexity and are not programmer friendly.

We present a simple and well-defined approach to obtain relevant results for shared memory environments through relaxing synchronization. For that, we will look into *Mergeable Data Types*, data structures analogous to *Conflict-Free Replicated Data Types* but designed to perform in shared memory.

CRDTs were the first formal approach engaging a solid theoretical study about *eventual consistency* on distributed systems, answering the CAP Theorem problem and providing high-availability. With CRDTs, updates are unsynchronized, and replicas eventually converge to a correct common state. However, CRDTs are not designed to perform in shared memory. In large-scale distributed systems the merge cost is negligible when compared to network mediated synchronization. Therefore, we have migrated the concept by developing the already existent Mergeable Data Types through formally defining a programming model that we named *Global-Local View*. Furthermore, we have created a portfolio of MDTs and demonstrated that in the appropriated scenarios we can largely benefit from the model.

RESUMO

É bem sabido que para garantir coerência em programas concorrentes num ambiente de memória partilhada sacrifica-se performance e escalabilidade. Alguns dos métodos existentes para garantirem resultados significativos introduzem uma elevada complexidade e não são práticos.

O nosso objetivo é o de garantir uma abordagem simples e bem definida de alcançar resultados notáveis em ambientes de memória partilhada, quando comparados com os métodos existentes, relaxando a coerência. Para tal, vamos analisar o conceito de*Mergeable Data Type*, estruturas análogas aos Conflict-Free Replicated Data Types mas concebidas para memória partilhada.

CRDTs foram a primeira abordagem a desenvolver um estudo formal sobre *eventual consistency*, respondendo ao problema descrito no CAP Theorem e garantindo elevada disponibilidade. Com CRDTs os updates não são síncronos e as réplicas convergem eventualmente para um estado correto e comum. No entanto, não foram concebidos para atuar em memória partilhada. Em sistemas distribuidos de larga escala o custo da operação de merge é negligenciável quando comparado com a sincronização global. Portanto, migramos o conceito desenvolvendo os já existentes Mergeable Data Type através da criação de uma formalização de um modelo de programação ao qual chamamos de *Global-Local View*. Além do mais, criamos um portfolio de MDTs e demonstramos que nos cenários apropriados podemos beneficiar largamente do modelo.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

1.1 CONTEXT

The CAP theorem [8] states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees: **Strong Consistency** [15], **Availability** and **Partition Tolerance**. Strong Consistency assures that every read returns the most recent write; Availability assures that every request receives a timely response; and Partition Tolerance assures that the system continues to operate despite arbitrary partitioning due to network failures. In other words, the CAP theorem states that in the presence of a network partition, we have to choose between Strong Consistency and Availability.

In order to provide high Availability, large-scale distributed systems often adopt a more relaxed consistency model called *eventual consistency* (EC). Replicating data under *eventual consistency* allows any replica to accept updates without remote synchronization, which ensures performance and scalability. With the exponential growth of peer-to-peer, edge computing, grid and cloud systems, EC has become an increasing topic of research in high-available, large-scale, asynchronous systems. *Conflict-free Replicated Data Types (CRDTs)* [33] was the first formal approach, grounded on simple mathematical properties, engaging a solid theoretical study about EC.

CRDTs are mergeable data structures where updates are unsynchronized but replicas still eventually converge to a correct common state. CRDTs remain responsive, available and scalable despite high network latency, faults, or disconnection. With these data structures we can achieve *strong eventual consistency* (SEC), which is a stronger convergence property, adding the safety guarantee that any two nodes that have received the same (unordered) set of updates, will converge to the same state. This is a good approach to ensure high-availability and also consistency in distributed systems.

In shared memory environments, synchronization is needed and the current methods are expensive. A good way to describe the behavior of concurrent objects is trough correctness and progress conditions. Linearizability is very useful to reason about the correctness of concurrent data structures but, implementations often lead to a bottleneck caused by

synchronization. Many applications apply ad-hoc techniques to eliminate the need of synchronous atomic updates, which may result in non-linearizable implementations.

Mergeable Data Types (MDTs)[4], are convergent data structures, used to relax synchronization in shared memory environments, analogously to CRDTs in distributed systems. The central idea is that each thread can update its local private copy of an object and later merge its changes to a shared global object. Each thread's execution will behave as a fork of the main branch (global shared object). At the moment that execution starts, each thread has a local copy of the global shared object, and when the merge operation performs, this new branch joins its updates into the main branch.

1.2 MOTIVATION

As the number of cores increases in a multi-core system, the synchronization cost becomes more apparent favouring the relaxation of concurrent object semantics for scaling the programs [34]. Programming patterns that attempt to limit the associated cost of the required synchronization on the memory accesses are emerging. As in CRDTs, the concept of mergeable data types is widely in use on distributed systems. Each replica can be concurrently updated, and later be merged with other replicas, while it is guaranteed that all nodes reach a convergent state once all updates have been delivered. Although, high latency network and possible reordering of messages in distributed systems, resulted in properties much different from what is required in shared memory systems. It is relatively simple to recognize that the use of such data types could be adapted, with enhanced and specific settings, to perform on a shared memory environment.

1.3 PROBLEM STATEMENT

Current shared memory solutions, use linearizability as correctness criteria under concurrent data structures because is useful to reason about the abstraction. On the other hand, is a prohibitively expensive criterion and going over it often leads to solutions that are hard to reason, verify and implement.

The applicability of mergeable data types in a multi-core shared memory setting has advantages and better performance could be achieved at the expense of linearizability. However, there is no consolidated theory on such data types in the shared memory ecosystem.

1.4 MAIN CONTRIBUTIONS

In this dissertation, we do an in-depth study and continue the work on *Highly-scalable concurrent objects* by *Deepthi Devaki Akkoorath and Annette Bieniusa* [4]. Mostly, we explore how

relaxing linearizability brings advantages to a shared memory environment by developing and formalizing MDT's previously proposed programming model. Part of the work described in this dissertation is published at *Global-Local View: Scalable Consistency for Concurrent Data Types* [5], presented at the conferences PMLDC 2017 and Euro-Par 2018. We will also present a portfolio with different data types and the respective implementations, comparing them with the linearizable ones. Part of this portfolio was delivered as artifact for Euro-Par where reviewers actually runned benchmarks over the MDTs.

1.5 DISSERTATION OUTLINE

The remaining chapters of this dissertation are organized as follows. Chapter 2 presents the current state of the art. Section 2.1 approaches linearizable and non-linearizable methods, in order to revisit the concepts and study examples of correctness conditions. We also contemplate some of the existing approaches to relax linearizability. Section 2.2 revisits the existing mechanisms to handle concurrency in shared memory environments. In Section 2.3 we see some programming models with resemblances to ours, where a per-thread replica is maintained to perform updates. In Section 2.4 we visit CRDTs and some important notions to retain.

In Chapter 3, we present the development of our work and an abstraction of our programming model. In Chapter 4 we present a portfolio of MDTs, and in Chapter 5 we present practical implementations of our portfolio establishing comparisons with the existing methods. Finally, in Chapter 6 we present some final considerations.

2

RELATED WORK

2.1 CORRECTNESS CONDITIONS

A correctness condition is a predicate on a history of a system [13]. Each history is a record of interactions between an object and its external interface calls. A method call is comprehended as the interval between its event *invocation* and the respective event *response*. When the call is waiting for a response, the event is described as *pending*. Concurrent histories may consist of both overlapping and non-overlapping calls. Correctness conditions are easier to reason about if concurrent executions are mapped into sequential ones. There are several well-known existing correctness conditions [20], and we will present some progress conditions that are often mentioned in the shared memory literature.

2.1.1 *Sequential Consistency*

Sequential consistency [25], or *serializable consistency*, is a simple condition requiring the order of method calls in a concrete history for a single process to be preserved. This is, method calls act as if they occurred in a sequential order consistent with program order.

Specifically, **calls should appear to happen in a one-at-a-time sequential order**. Let's say we have two concurrent writes A and B on a register holding a value, where a write operation assigns the value on the register, and a read operation performs a query. In the presence of concurrent writes the later one overwrites the earlier one. So if we have the concurrent writes A and B at the end we should not have unexpected results, meaning that either we have A or B, and not a mixture between them. Also, **calls should appear to take effect in program order**, meaning that, the order of each thread's calls should be preserved. If we have a sequential execution with two writes A and B, where A takes effect and B takes effect after A, a following read should display the effect of B.

Calls performed by different threads may be reordered in the history even if the operation calls do not overlap in the execution. With sequential consistency, we have local order with arbitrary interleaving between threads executions. Sequential consistency is also not com-

posable, the result of composing multiple sequentially consistent objects it is not sequential consistent by itself, as shown in the next example:

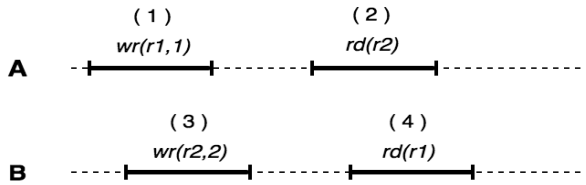


Figure 1: Writing and reading from two different registers: r1 and r2

Given the register semantics, Figure 2 represents the possible ordering of operations that result in a sequentially consistent execution:

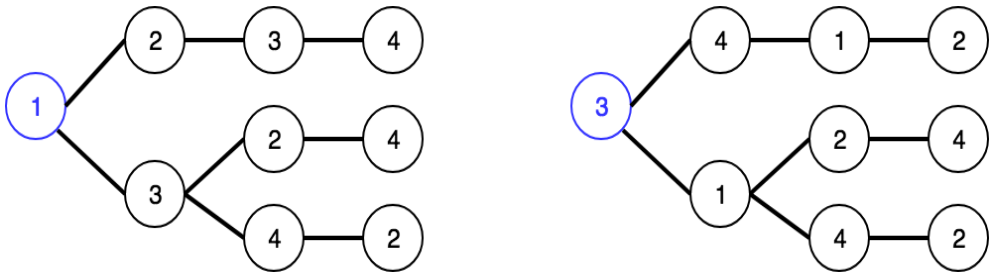


Figure 2: Sequentially consistent operation ordering from Figure 1

Assuming that both *r1* and *r2* are initialized with value 0. From thread's A and B executions we have four different possible local scenarios:

- $A1 : rd(r2) = 0$. Justified by the order of operations: (1), (2), (3), (4);
- $A2 : rd(r2) = 2$. Justified by any ordering on Figure 2 where (3) happens before (2);
- $B1 : rd(r1) = 0$. Justified by the order: (3), (4), (1), (2);
- $B2 : rd(r1) = 1$. Justified by any ordering on Figure 2 where (1) happens before (4);

All these scenarios are independently sequentially consistent but the composition of any two scenarios is not necessarily sequentially consistent. $A1$ followed by $B2$ is sequentially consistent, justified by (1), (2), (3), (4); $B1$ followed by $A2$ is also sequentially consistent by the order (3), (4), (1), (2); but the same is not true for the composition of $A1$ with $B1$ because there is no operation ordering, respecting the register semantics, where $rd(r2) = 0$ and $rd(r1) = 0$.

2.1.2 *Linearizability*

Linearizability [21] has turned out to be a fundamental notion on simplifying the reasoning about the correctness of shared data structures for programmers. It is a real-time guarantee on the behavior of reads and writes on single objects.

Each method call should appear to take effect instantaneously at some moment between its invocation and response. In an execution, every method call is associated with a linearization point, a point in time between its invocation and its response. The call appears to occur instantaneously at its linearization point, behaving as specified by the sequential definition. Informally, writes should appear to be instantaneous, once a write completes, all later reads should return the value of that write or the value of a later write. For instance, in a lock-base implementation, critical sections can serve as linearization points. On other implementations, linearization points are typically a single step where the effects of the call became visible to other methods.

This correctness criteria formalizes the notion of atomicity for high-level operations and is the strong consistency(C) in *Gilbert and Lynch's* conjecture of the CAP Theorem [15].

Linearizability is composable because, if operations on each object at a system are linearizable, then all operations at the system are linearizable. This is an important property for large and complex systems because it favors a modular implementation of the different parts of a system, making them proved correct independently.

With linearizability we have global order, being a more restrictive criterion than sequential consistency. Stronger restrictions yield stronger consistency. Under linearizability, we have sequential consistency but the reverse is not true.

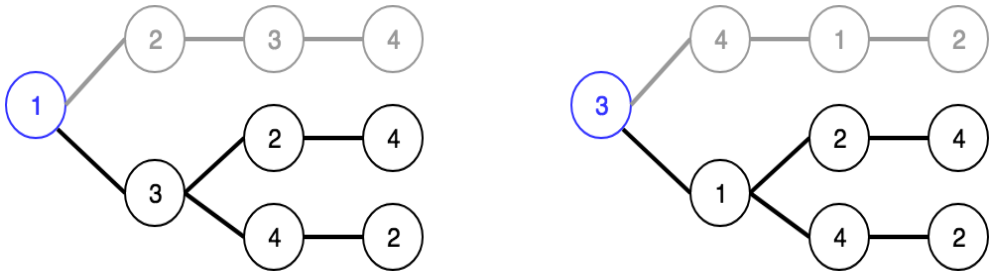


Figure 3: Linearizable opration ordering from Figure 2

Following Figure 2, in Figure 3 we have highlighted, from all the sequentially consistent operation ordering, the ones that are also linearizable and grayed out the ones that are only sequentially consistent. As we can observe, all linearizable executions are sequential consistent but the other way around is not true.

2.1.3 *Serializability*

Serializability is a **guarantee about transactions**, a group of one or more operations over one or more objects, which ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. **An execution is serializable if we can find a serial execution equivalent to it**. Serializability is the isolation(I) in the ACID properties, also know as *serializable isolation* and is seen as a mechanism for guaranteeing database correction.

The concept does not impose constraints in transaction ordering, meaning that transactions could be reordered. It ensures that transactions execute as if they were executing one at a time on a single copy of the database. It is also not composable.

Although the concept has an origin in database systems, the principles tend to converge with the principles of sequential consistency and are often referred as the same. We will treat them as separated concepts. Linearizability is also both present on the terminology of distributed, concurrent, and database systems. It could be seen as a special case of strict serializability [21] where transactions are restricted to consist of a single operation applied to a single object.

2.1.4 *Other Relaxed Models*

Some models attempt to relax the strict semantics of linearizability in order to achieve better performance.

Quiescent consistency[7] provides high-performance at the expense of weaker constraints satisfied by the system. As in sequential consistency, method calls should appear to happen in a one-at-a-time sequential order. Moreover, method calls separated by a period of quiescence should execute in their real-time order. An object is in a *quiescent state* if currently there is no pending or executing operation on that object

K-linearizability[23] is a criterion between sequential consistency and linearizability allowing to reorder arbitrarily up to $k-1$ ($k > 0$) sequential operations and thus execute concurrently. This is a relaxed criteria since linearizability only allows to reorder concurrent operations and not sequential.

Quasi-Linearizability[2] allows each operation to be linearized at a different point at some bounded distance from its linearization point. For instance, a queue that dequeues in random order but never returns empty if the queue non-empty is a quasi-linearizable queue. Allows more parallelism by allowing flexible implementations.

Weak and Medium Future Linearizability [24] is a consistency model applicable to the data types implemented using futures, which allows flexible reordering of the operations.

2.1.5 *Summary*

Correctness criteria are hard to explain and the terminology of distributed, concurrent and database systems is not consistent.

Sequential consistency is a good way to describe standalone systems, such as hardware memories, where composition is not an issue. Serializability is a mechanism to ensure database correctness. Linearizability strengths sequential consistency by requiring the order of non-overlapping operations to be preserved and could be seen as a special case of serializability. It is a good and simple way to describe components of large systems, where components must be implemented and verified independently. Because linearizability is also expensive, due to the imposed constraints, weaker criteria started to gain form.

2.2 CONCURRENCY MECHANISMS

In concurrency control for shared memory systems, we concern about the correct and efficient execution of concurrent operations. Across the time several works have been developed in this field, and there is no standard or conventional approach, mainly due to the large variety and diversity of solutions. Also, each one has its own trade-offs, and we can only state that some approaches are better suitable for specific scenarios than others.

One of the most primitive and popular mechanisms for inter-process synchronization are mutual-exclusion locks, although their misleading simplicity explains the widespread. In a context of high-performance applications, more and more efficient concurrency control is required and, it is well proved that mutual exclusion locks don't scale with large numbers of locks and many concurrent threads. Scalability can be captured by a measure known as the *speedup* of the implementation. This measure effectively characterizes how an application is using the machine it is running on. On a machine with P processors, the *speedup* is the ratio of the execution time on a single processor(*T(1)*) to its execution time on P processors(*T(P)*).

$$speedup = \frac{T(1)}{T(P)}$$

Data structures whose *speedup* grows with P are called *scalable*.

Highly-concurrent access to shared data demands a complex, fine-grained locking strategy such that non-conflicting operations can execute concurrently. It is a rough path to design correctly and efficiently such strategies because they carry lots of problems that need to be avoided. *Priority inversion* occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. *Convoying* happens when a thread holding a lock is descheduled due to a time-slice interrupt or page fault. *Deadlock* can occur

if threads attempt to lock the same objects in different orders. Avoiding *deadlocks* is difficult if threads must lock many objects, particularly if the set of objects is not known in advance. With these known problems, interest in solutions like *non-blocking* mechanisms grew. It was expected that efficient and robust systems could be built, but the reality is that most algorithms are complex, slow and impractical.

Nowadays it is infeasible for programmers to build large systems using these primitive methods. Therefore, more evolved solutions appeared. In this section, we present, Concurrent Data Structures Libraries and Transactional Memory, two popular approaches in concurrency control for shared memory systems.

2.2.1 *Concurrent Data Structures Libraries (CDSLs)*

This first approach is to use libraries with data structures that already handle concurrency. *Concurrent data structures* (CDSs) are data structures intended for use in parallel environments, also called "thread-safe". By "thread-safe", we interpret that each data structure operation executes atomically and in isolation from other operations on the same data structure. In this scenario, the correctness criteria generally required is *linearizability*.

The fast adoption of commercial multiprocessor machines has brought about significant changes in the art of concurrent programming and drew attention to data structures. The efficiency of these data structures is crucial to performance, yet designing concurrent data structures is an art currently mastered by few. CDSs are far more complicated to design than sequential ones because threads executing concurrently can interleave their steps in many ways, producing unexpected outcomes. Threads execute concurrently in different processors and operational systems, therefore, can be subject to SO interventions which difficult concurrency. Also, the drive to improve performance often makes it harder to design and verify the correctness of a CDS implementation. In fact, when designing CDSs the use of the primitives presented before is unavoidable, ending up carrying the same problems.

We will use, a simple example of a shared counter presented by *Mark Moir and Nir Shavit* [31] that illustrates some difficulties of the CDS design and some possible improvement techniques. This counter has a unique operation **fetch-and-add**, implemented with mutual-exclusion locks:

```
1 acquire(Lock);
2 oldval=X;
3 X=oldval+1;
4 relase(Lock);
5 return(oldval);
```

This naive implementation shows that the careless application of this mechanism introduces three problems: a sequential bottleneck because at any point at most one **fetch-and-add** operation is executing, an overhead in traffic in the underlying hardware as a result of multiple threads concurrently attempting to access the same locations in memory(*memory contention*) and a phenom called *blocking* that occurs when the thread that holds the lock is delayed. The problems could be worked by using a fine-grained locking scheme, i.e., use multiples locks to protect smaller pieces of the code, allowing concurrent operations to proceed when they're not accessing the same parts of the data structure. However, fine-grained locking doesn't scale. One alternative is to use a sophisticated technique known as *combining tree* to implement a scalable counter. This method uses a binary search tree where each leaf is a thread, and the root contains the actual value. If a thread wants to increment the counter, it starts from the leaf node and works its way up the tree to the root, using the intermediate nodes to combine with other threads. When two threads combine at one node, one is called *loser*, and another *winner*. The *loser* simply waits at the node until a value is delivered from the *winner* and the *winner* climbs the tree carrying the sum of the values of his sub-tree.

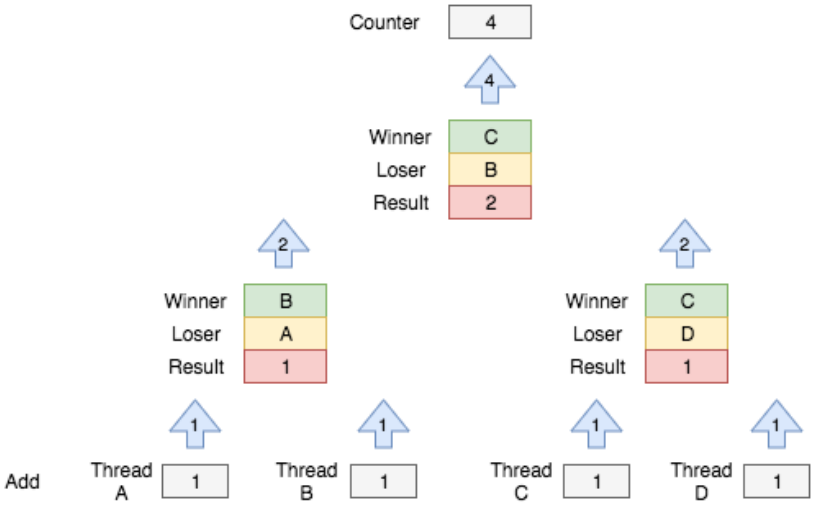


Figure 4: Combining tree upstream process

When a thread reaches the root of the tree, adds its sum to the counter in a single addition, applying the effects of all the combined operations and fetches the old value passing this one down the tree. Thus, if multiple threads are increasing the counter at approximately the same time, the maximum number of threads that will compete for accessing the counter variable is two, no matter how many threads are making increment requests.

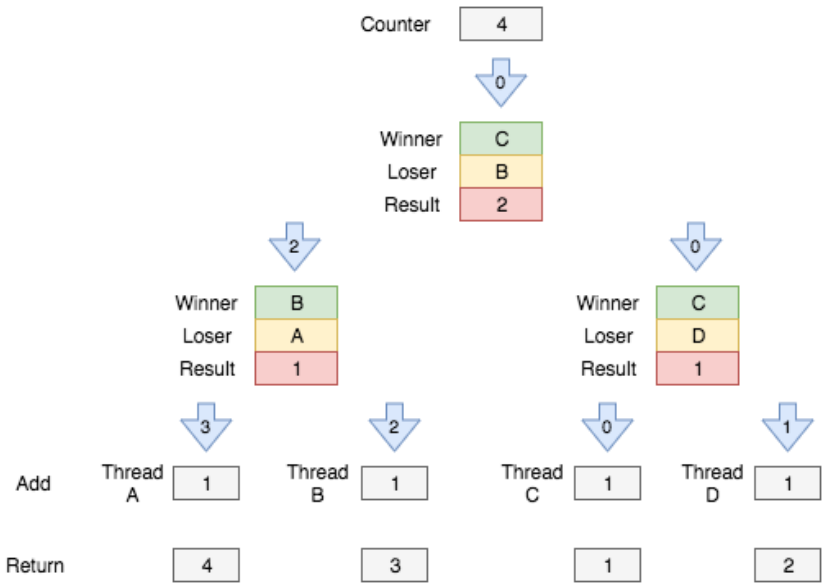


Figure 5: Combining tree downstream process

Combining tree reduces contention for individual memory locations, reduces memory traffic and allows operations to proceed in parallel when comparing to the previous single lock version. However, this technique as some disadvantages: it requires a known bound of the number of threads that access the counter; when accessed by many concurrent threads under low loads the best case is poor ($O(logN)$) when compared to the constant time in the previous version; if a thread fails to combine because it arrived at a node immediately after a winner left it on its way up the tree, then it must wait until the winner returns before it can continue its own ascent up the tree; coordination is complex and may lead to *deadlocks*, and avoiding *deadlocks* complicates the design task.

Non-blocking techniques overcome some of the problems presented before related to the use of locks. Two *non-blocking* progress conditions should be mentioned: *wait-free* and *lock-free*. A *lock-free* operation ensures at least some thread is making progress on its work (*system-wide progress*). After a finite number of its own steps, some operation completes. A *wait-free* operation ensures that any thread will be able to make some progress and eventually complete (*system-wide and per-thread progress*). After a finite number of its own steps the operation completes, regardless of the timing behavior of other operations. This means that an operation that is said to be *wait-free* gives the same (and more) progress guarantees than a function that is *lock-free*. *Mark Moir and Nir Shavit* [31] state that there is no manner to implement a shared counter in a *lock-free* or *wait-free* way using only *load* and *store* as separated instructions. Therefore, they propose to use a synchronization instruction present in most modern processors called *compare-and-swap*(CAS) to guarantee that a labeled piece of code executes atomically.

Furthermore, with the CAS operation, we can perform our **fetch-and-add** with a *wait-free* implementation because it atomically loads from a memory location, compares the value read to an expected value, and stores a new value to the location if the comparison succeeds. It is also possible to have only a *lock-free* implementation by loading the counter's value and then use CAS to atomically change the counter value to a value greater by one than the value read. The CAS instruction fails to increment the counter only if it changes between the load and the CAS. When it fails, the operation can retry, and the failing one has no effect. It only fails as a result of another **fetch-and-add** operation succeeding. Although this implementation is *lock-free* it is not *wait-free* because the CAS can repeatedly fail. Also, it has some disadvantages of the previous single-lock version: the sequential bottleneck because at most one operation can perform and high contention for a single location. If we keep the same line of thought as in the single-lock version, we should try to apply some improvements to the *lock-free* version, but generally, improvements lead to expensive and more complicated techniques because the absence of locks make us concern about providing a correct implementation, despite actions of concurrent operations, instead of preventing that other threads interfere with the work that some thread is doing.

This example clearly illustrates that the design of concurrent data structures provides significant challenges, by far a lot more complicated than sequential ones. As we surpass some challenges, new ones arise and we can end up sacrificing performance. On the other hand, well-designed concurrent data structures can achieve notable performance results, but for that, it is necessary to achieve a certain maturity in the design. The use of libraries with data structures that serve our purpose, and where concurrency is optimized in a high-level, is a comfortable solution although the lack of understanding of the implementation details is not always accepted. In the next chapter, we present a transactional approach that makes it much easier and understandable to design systems for shared memory environments.

2.2.2 *Transactional Memory(TM)*

In the previous section, we have seen that the fundamentals in concurrent data structures design aim to modify multiple memory locations atomically so that no thread can observe partial results from other threads. With transactional mechanisms, we can also treat sections of code that access multiple locations as a single atomic step, which simplifies the design of data structures in the sense that we don't have to concern about the problems that are inherent to the use of the previous mechanisms.

A *transaction* is a sequence of steps executed by a single thread. *Herlihy and Shavit* [20] state that transactions should be *serializable*. One way to implement *serializability* is to use *speculative transactions*, transactions that make tentatives changes to objects. A transaction without synchronization conflicts commits and the tentative changes become permanent. A

transaction with synchronization conflicts aborts and the tentative changes are discarded without the need of concern about deadlocks.

Transactional memory can be implemented in either hardware or software. A hardware approach may have high performance, but imposes strict limits on the amount of data updated in each transaction. A software approach removes these limits but incurs high overhead.

Hardware Transactional Memory(HTM)

Transactional support for multiprocessor synchronization was first introduced by *Herlihy and Moss* [19], allowing shared-memory operations to be grouped into atomic transactions. A hardware-based transactional memory mechanism motivated by the *cache-coherency* protocols present in modern multiprocessors was also proposed. These protocols already do most of the work needed to implement transactions and as they are implemented in hardware, they perform very efficiently. Specifically, in modern multiprocessors, each processor stores a copy of shared data in a small high-speed *cache* used to avoid communication with another copy of shared data present in the processor slow main memory. Each cache entry holds a group of neighbor memory addresses called *line*. Cache lines with the same memory addresses in different processors are synchronized through the manipulation of a simple state machine. Simply speaking, this approach keeps track of whether a processor tried to modify a memory address that another has in its cache. The hardware keeps track of which cache lines have been read from and which have been written to. If two processors read the same cache line addresses, there's no conflict. If either writes to a cache line that another has read or written to, then there's a conflict, and one of the transactions will fail. This allows us to detect and resolve synchronization conflicts to shared memory between writes, and between reads and writes. Also, tentative changes are buffered instead of updating memory directly. *Herlihy and Moss* with small changes to these protocols proposed their HTM but unfortunately, the approach was naive.

When a transaction fails the most apparently obvious thing to do is to restart it, but it is not so simple. This protocol doesn't guarantee progress, unless we ensure that at least one transaction will succeed what is difficult because transactions might fail, for instance, due to interrupts. It is also difficult, to respect thread priorities. High-priority threads could be starving while lower-priority threads with shorter transactions always succeed. However, the major problem with this approach is that hardware transactions have inherent capacity constraints on the number of distinct memory locations that can be accessed inside a single transaction because the size of the cache limits the size of a transaction.

Then, it is not viable to design HTM systems since to surpass these problems it would require hardware modifications, which requires that processor manufacturers concern about HTM. For many years, efforts to take advantage of the benefits from hardware-transactional

synchronization were made, combining hardware and software transactional memory implementations, also known as *hybrid transactional memory*(HyTM), like the ones presented on [11] or [29]. However, hybrid transactional memory solutions suffer from a significant drawback: the coordination with the software introduces an unacceptably high instrumentation overhead into the hardware transactions. Intel introduced the concept of *Reduced Transactional Memory* (RTM), developing their *Haswell microarchitecture*, which allowed to eliminate most of the instrumentation.

Software Transactional Memory(STM)

A purely software-based transactional memory was proposed by *Shavit and Touitou* [35] addressing the problems of *Herlihy and Moss* hardware transactional memory. This STM was *non-blocking*, in contrast with the *blocking* previous HTM, and is implementable on the existing hardware. A notable feature is that they abort contending transactions rather than recursively helping them. Their design supported only *static* transactions, that is, transactions which access a pre-determined sequence of locations. Although this approach couldn't aim for the same overall performance than the HTM, some advantages are identified: portability among machines and the tolerance on the presence of timing anomalies and processor failures. In fact, this proposal was the first step for the beginning of transactional memory widespread. Later, *Moir* presented an STM design [30] for converting sequential implementations of dynamic transactions into non-blocking implementations, allowing *dynamic* programming in contrast with *Herlihy and Moss* static approach. Following, various forms of software transactional memory have been proposed like *Herlihy et al.* [18] or *Harris and Fraser* [16].

The design of transactional memory trends to provide large-scale transactions. Large-scale transactions are achieved by assuring that transactions are *unbounded* and *dynamic*: *unbounded* means that there is no limit on the number of locations accessed by a transaction, and *dynamic* means that the set of locations accessed by the transaction is not known in advance and is determined during its execution. *Dice et al.* [12] presented an STM approach called *Transactional Locking II*, with *dynamic* and *bounded* transactions, which is based on a combination of commit-time locking and a global version-clock based validation technique, allowing dynamic and unbounded transactions. Although, performance kept being a problem mainly due to coarse-grain *conflict detection*, which limits parallelism producing unnecessary conflicts and high abort rates. Transaction memory systems synchronization is on the basis of *read/write conflicts*. As a transaction executes, it records the locations it reads in a *read-set* and the locations it wrote in a *write-set*. Conflicts appear if either a transaction read-set or write-set intersects other transaction write-set. Although this kind of synchronization could happen without the programmer participation, it can severely and

unnecessarily restrict concurrency for certain shared objects that could be "*hot-spots*" of the system, ending up with significant losses of performance.

Let's take as an example a simple sorted sequence implemented by a *linked-list*. Consider that each node of the list holds a unique integer *value* and a *reference* to a next node. The nodes are sorted by value and the operation add(x) is provided, where x is the value of a new node added to the list. If x is absent, nodes are traversed until it is found x'*s successor*(a value greater than x) to insert the new node. This new node reference points to x successor, and the *predecessor*(the previous node) points to x. Consider the list initial state of: {1,3,5,6,7,9}. And, two pending concurrent transactions: Transaction A that performs add(4) and Transaction B that performs add(8). Neither of the transactions calls depends on each other, but as illustrated in Figure 6 they conflict.

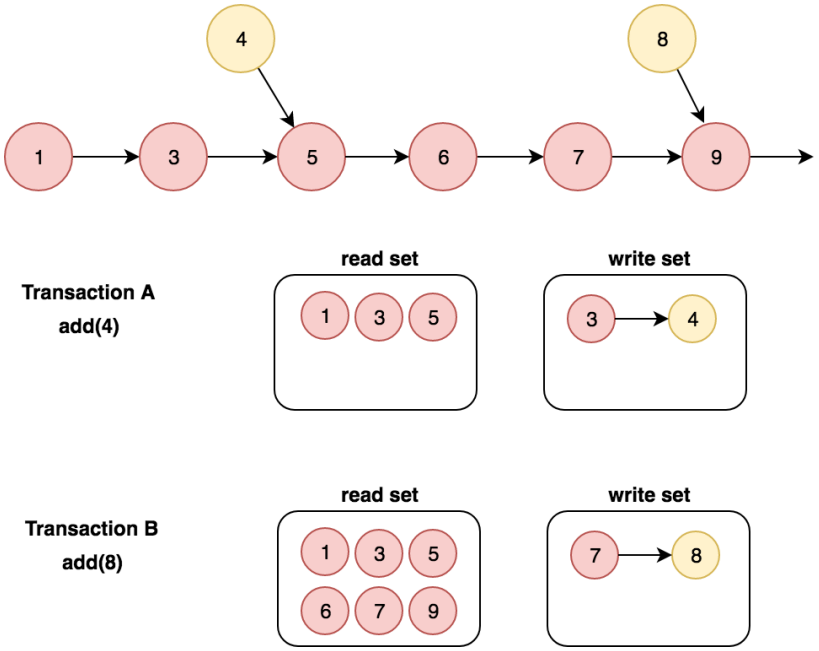


Figure 6: TM conflict detection

Conflicts are caused because the transaction's steps interleave. Node 3 is on transaction's A write-set and at transaction's B read-set, causing a conflict. Although adding node 3 would not affect adding node 8, the way traditional TM detect conflict produces the unnecessary abort from one of the transactions.

In the next section, we will see that some more recent approaches tried to solve the STM efficiency problem by combining "the best of both worlds": The efficiency of concurrent data structures with transactional memory more friendly implementation.

2.2.3 *Combining the two approaches*

Addressing the problem of restricted concurrency on software transactional memory *Herlihy and Koskinen* [17] proposed *transactional boosting*, a methodology to convert concurrent data structures, implemented with lock or lock-free mechanisms, into transactional ones. Using the semantics of well-defined data structures is possible to perform better conflict detection on concurrent transactions, allowing the reduction of abort rates significantly and consequently better concurrency. *Afek et al.* [1] showed how to enhance an STM, presenting an approach called *consistency oblivious programming* (COP) which allows executing sections of code that meet certain conditions without checking for consistency in order to improve parallelism. As we have seen before, to provide conflict detection most transactional memory systems keep track of the read and write sets of transactions. In most STMs, this instrumentation of threads interactions is expensively done with *word-based bookkeeping*, i.e., keeping track of every memory address that transactions access, results in poor scalability. COP tries to avoid that, using sequential code from data structures to see if operations can optimistically execute without keeping track of any transaction interactions. If they can't execute, COP just retires to execute them in a more expensive way. *Herman et al.* [22] presented an STM, called *software transactional objects* (STO), which tracks abstract operations over data structures instead of the read and write sets. When compared to the conventional STMs, this STO reduces bookkeeping, limits false conflicts and implements an efficient concurrency control. The last three examples, use concurrent data structures to improve transactional memory systems. *Spiegelman et al.* [36] in *Transactional Data Structures Libraries* (TDSLs), bring transactions into *concurrent data structures libraries* (CDSLs), so that no performance is sacrificed and sequences of operations in a data structure could be atomic. Their library offers stand-alone operations at the speed of a custom-tailored CDSLs, and at the same time, provides efficient and scalable transactions that are way faster than conventional software transactional memory systems.

2.2.4 *Summary*

Primitive methods for concurrency control, like mutual-exclusion locks, are not viable to build large-scale concurrent systems. CDSLs are an efficient solution if concurrent data structures are properly implemented, although their design is hard. Hardware transactional memory promised to reduce the difficulty of writing correct, efficient, and scalable concurrent programs but unfortunately, there is no viable support of hardware. Over the years purely software alternatives appeared but the efficiency falls short. Recently, appeared solutions that combine transactional memory with concurrent data structures taking the best of both.

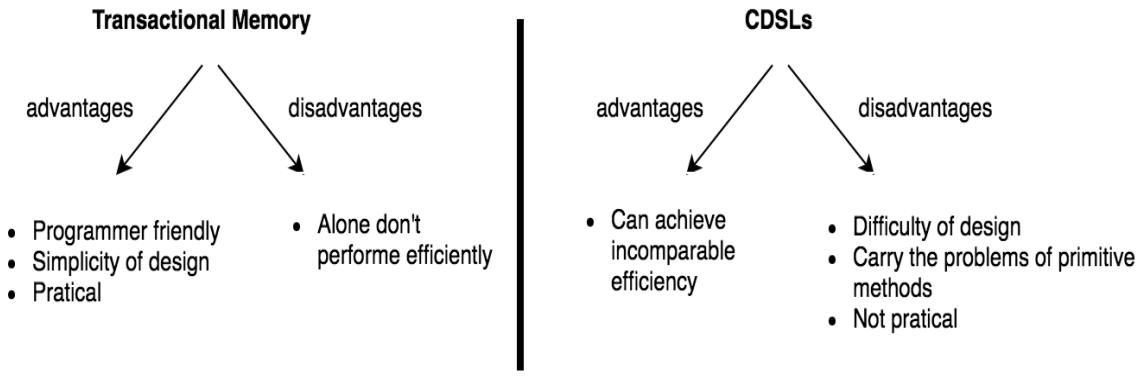


Figure 7: Transactional Memory vs CDSLs

Our goal with MDTs, is to provide the advantages that these two approaches can only provide together in a simple and understandable way.

2.3 Related Programming Models

Maintaining per-thread replicas and performing updates on them has been considered by different programming models in the literature.

2.3.1 *Concurrent Revisions*

In *Concurrent Programming with Revisions and Isolation Types*[9] a programming model is presented to simplify the sharing of data between threads. The general idea is the development of the fork-join model to suit this purpose. A forked thread's state is initially, a copy of its parent state. The forked thread makes changes on its copy, and these changes are merged to the parent thread when it is joined back. During the join, conflicting updates are resolved using type-specific merge operations. Specifically, two important notions are defined: *isolation types* and *revisions*.

Isolation types are special types used to declare what data can be shared between threads. Mainly two categories of isolation types were identified. *Versioned* types that are suitable in situations where concurrent modifications happen with low frequency, or when there is clearly priority between threads. For instance, joinee thread's effects always override. *Commulative* types are useful in situations where the results from different threads should be combined, and the final result is achieved through a merge function.

Revisions are tasks that are forked and joined, and could themselves be forked. Similar to branches in source control systems. Each revision read and modify it is own private copy of the entire shared state, with the guarantee to be consistent, isolated and stable. There is

no need to perform any synchronization inside revisions, providing simplicity of use. Also, reason about how threads may see or not each other effects is simplified by *revision diagrams* (Figure 8).

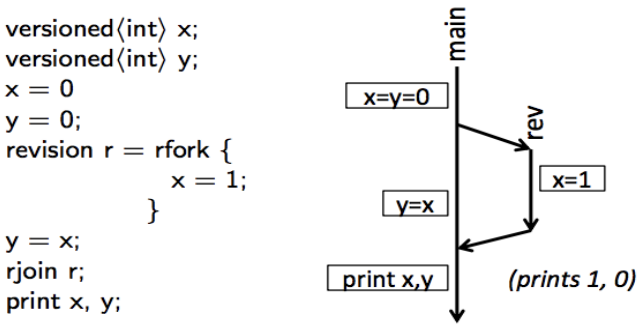


Figure 8: Example of Revision Diagram

The focus of this work is on the fork-join model, where threads can communicate their state only when they join their parent. In contrast, we provide a generic model for data types where a two-way merge and strong updates can share state among the threads at any point in the execution, thus enabling applications to tune their use.

2.3.2 *Global Sequence Protocol(GSP)*

Global Sequence Protocol [10] is a model for replicated and distributed systems. Its used to describe the system behavior precisely, yet abstractly enough to be suitable as a simple mental model that can be instantiated to any data type. A data type is abstracted by a set of *Read* operations, a set of *Update* operations, and a set of *Values* returned by read operations. The abstract semantics of operations is represented by a function *rvalue* that receives a read operation and a sequence of updates. This operation returns the value that results from applying all of the updates in the sequence to the initial state of the data.

Observing the behavior of programs as a sequence of updates, and not states, is enforced. A state is the sequence of updates that have led to it. GSP has a global state which is represented as a sequence of operations. Each client stores a prefix of this global sequence. It is based on the simple idea that clients eventually agree on a global sequence of updates while seeing a sub-sequence of the final sequence at any given point of time. For instance, a counter could be abstractly defined as follows:

```
1 Update = { inc }
2 Read = { rd }
3 rvalue (rd,s) = s.length
```

The above snippet represents a simple read counting the number of updates(increments) on the sequence of updates *s*.

On a distributed system, the updates are first appended to a local sequence of pending operations and then broadcasted to other replicas using a reliable total order protocol which enforces a single order on the global sequence. There is much less control on replica divergence and *liveness* of the global sequence evolution because there are no bounds on message delays. Liveness[20] is a system property that states that a particular "good" thing will happen. For instance, a red traffic light will eventually turn green.

In contrast, we address a shared memory concurrent architecture that allows to reason about bounds on divergence and stronger progress guarantees on the evolution of the shared state.

2.3.3 *Read-Copy-Update and Read-Log-Update*

Read-copy-update (RCU) [28] is a synchronization mechanism that allows threads to read a shared object while a concurrent modification is in progress. This mechanism is widely in use at Linux kernel since 2002 [27], bringing significant scalability improvements.

Similar to our model multiple versions of an object are maintained, so that, reads observe a consistent state while a modification is in progress. The core idea of RCU is to duplicate an object each time it is modified and perform the modifications on the private copy. Reads access the unmodified data structure while updates modify the copy. Once modifications are complete, they are installed using a single pointer modification in a way that does not interfere with ongoing readers. To avoid synchronization, updates wait until all pre-existing readers have finished their operations, and only then install the modified copies. To ensure consistency, such updates must be done at a *safe point* when no reader can potentially hold a reference to the old data. One important property of RCU is the ability to determine this safe point. When all threads have passed through a *quiescent state* since a particular point in time, they are all guaranteed to see the effects of any change made before the beginning of the interval. This guarantee allows many algorithms to be constructed using fewer locks or even no locks at all.

RCU defines and uses efficient and scalable mechanisms for publishing and reading new versions of an object, and also for garbage-collecting old versions. These mechanisms distribute the work among reads and updates in such a way that makes reads extremely fast. Although introducing significant improvements while avoiding synchronization, RCU it is only suitable for a scenario where we have *multiple readers and only a writer*. Also, threads experience delays when waiting for readers to complete their operations. Besides,

some crucial aspects as synchronization between writes and the duplication of objects are not delivered out of the box by RCU and need to be concerned at implementation time.

Read-log-update (RLU) [26] is an extension over RCU to solve the problems presented before. The RLU core is a logging coordination mechanism inspired by transactional memory algorithms. It provides support for *multiple writes in a single operation* by combining RCU ability to determine the safe point to perform writes with clock-based logging mechanism that replaces the manual creation and installation of objects.

RLU maintains a per-thread write-log where write operations log each object they modify. Objects are automatically copied into the write-log and manipulated there. Also, RLU maintains a global-clock, which is a memory location that is updated by threads when they wish to establish a synchronization point. This global clock is read at the start of every operation and is used as a timestamp for thread's operations.

Each writer thread, after completing the modifications to the logged object copies, commits them by incrementing the clock to a new value. At this point, operations are divided between old and new operations. Old are the ones that started before the clock increments and will read old object copies. The new ones started after the clock increment. Then, the writer waits for old operations(reads) to finish, as in RCU, and after that writes the objects in the writer log to memory using locks, overriding the old objects. Meanwhile, new operations started using the ongoing write-log without waiting for objects to be written in memory.

In contrast with our model, on RCU and RLU concurrent writes are sequential consistent which is achieved by sequentially order the writes or by fine-grained locking.

2.3.4 *Summary*

In this section, we visited four programming models that perform updates on per-thread replicas. Concurrent Revisions is an expansion of the fork-join model where forks can do work and later merge it to their parent. Global Sequence Protocol is an abstraction that looks into shared memory programs state as sequences of updates, allowing threads to work seeing only sub-sequences of these updates and later append the updates on the global sequence. Read-Copy-Update and Read-Log-Update maintain shared and thread local representation of objects to provide scalability enabling multiple-readers/one-writer and multiple-readers/multiple-writers respectively.

2.4 Conflict-free replicated data types (CRDTs)

MDTs use concepts from analogous data structures for replicated distributed systems, called *Conflict-Free Replicated Data Types* [33]. CRDTs were the first formal approach, grounded on

simple mathematical properties, engaging a solid theoretical study about *eventual consistency*, answering to the *CAP Theorem* problem. With CRDTs, updates are unsynchronized, and replicas eventually converge to a correct common state. CRDTs remain responsive, available and scalable despite high network latency, faults, or disconnection.

In this chapter, we will visit some important concepts, and some work with CRDTs that we consider relevant. There are two types of CRDTs: Convergent Replicated Data Types (CvRDTs) and Commutative Replicated Data Types (CmRDTs). Also, we will present some examples from the CRDTs portfolio that later we will use with MDTs.

2.4.1 *CAP Theorem and Strong Eventual Consistency*

The *CAP Theorem* [8] states that it is impossible in a distributed system to provide, simultaneously **consistency**, **availability** and **partition-tolerance**. *Availability* assures that every request receives a response. *Partition-tolerance* assures that the system continues to operate despite arbitrary partitioning due to network failures. *Consistency* assures that every read returns the most recent write, meaning that every node is up to date and in the same state. Actually, there are several forms of consistency. This one, linearizability, is known as *strong consistency* [15] and is expensive and hard to achieve. In the last years, weaker consistency forms, as *eventual consistency* started to gain some attention.

Eventual consistency (EC) seeks to ensure that replicas of a same mutable shared object converge without synchronization in the first plan. It is not guaranteed that nodes are always synchronized, but eventually, they will converge and be consistent. In *Conflict-Free Replicated Data Types*, EC intuition is captured by: *eventual delivery*, *convergence* and *termination*.

Eventual delivery: *An update delivered at some correct replica is eventually delivered to all correct replicas.*

Convergence: *Correct replicas that have delivered the same updates eventually reach equivalent state.*

We say that two states s and s' are *equivalent* if all queries return the same result for s and s'.

Termination: *All method executions terminate.*

The *eventual consistency* model sacrifices consistency to provide both availability and partition-tolerance. However, executing operations without coordination between replicas can originate conflicts that must be resolved. In order to be able to apply updates immediately without later conflicts, CRDTs satisfy a stronger property than EC, called *strong eventual consistency* (SEC). If we have EC to achieve SEC we need to add *strong convergence*:

Strong convergence: *Correct replicas that have delivered the same updates have equivalent state.*

In CRDTs updates are *commutative*, meaning that can be applied in any order. An update has two phases: a first one, a client performs a query in one of the replicas, designed *source*; and a second one, called *downstream* phase, where the update is transmitted asynchronously to all replicas. In the next section, we present *state-based* CRDTs or *Convergent Replicated Data Types* (CvRDTs).

2.4.2 *Convergent Replicated Data Types (State-based)*

In state-based, replication is passive, and an update occurs entirely at the source. In the *downstream* phase, updates are propagated transmitting the modified *payload* between arbitrary pairs of replicas. The *payload* could be the full state of the object or a delta of the state [6]. There is no need to know all the replicas involved, updates are propagated between arbitrary pairs of replicas. When a replica receives a payload, updates his payload with the output of the *merge* operation invoked with two arguments, the received state, and his payload: **merge** (*value1*, *value2*) : *payload mergedValue*.

State-based objects take their payload values from a *join semilattice*, defined as follow:

Join semilattice: *An ordered set* $(S, \leq_v)$ *is a join semilattice iff:* $\forall x, y \in S, x \sqcup_v y$ *exists.*

Least Upper Bound: $m = x \sqcup_v y$ *is a least upper bound of* $\{x, y\}$ *under* $\leq_v$ *iff:* $x \leq_v m$ *and* $y \leq_v m$ *and there is no* $m' \leq_v m$ *such that* $x \leq_v m'$ *and* $y \leq_v m'$.

Partial order ($\leq_v$): *A binary relation* $\leq_v$ *on a set S, is said to be a partial order if* $\forall x, y, z \in S$ *the following properties are satisfied:*

- $x \leq_v x$ ***(reflexive property)***
- *if* $x \leq_v y$ *and* $y \leq_v x$ *then x=y* ***(antisymmetric property)***
- *if* $x \leq_v y$ *and* $y \leq_v z$ *then* $x \leq_v z$ ***(transitive property)***

The pair (S,$\leq_v$*) is called partially ordered set.*

The definition of the operation *compare*(x,y) gives us the partial order definition $x \leq_v y$. By the *antisymmetric property*, two states are said equal if $x \leq_v y$ and $y \leq_v x$. The *merge* operation converges towards the LUB of the replica payload and the received payload: $merge(x, y) = x \sqcup_v y$. By the properties of LUB, *merge* is *commutative* and *idempotent*, which allow us to receive updates out of order and receive the same update multiples times without problem. In order to assure that messages can be lost, we need to guarantee that the payload after an update is greater or equal that the one before ($\leq_v$), this is know as a *monotonic join semilattice*. This way, recent updates, carry all the modifications in previous updates.

Counters

A Counter is a replicated integer supporting the operations *increment* and *decrement* for updates, and the operation *value* for reads. The idea is that the value of the counter converges with the global number of increments minus the number of decrements. As we can see increments and decrements commute.

P-COUNTER A **P-Counter** is a counter that only allows the *increment* operation. For illustration purposes, we will use a state-based approach presented in **Specification** 1.

Specification 1: State-based increment only-counter (vector version)

```
1  payload integer[n]P
2      initial [0,0,...,0] ;
3  update increment ()
4      let g = myId() ;
5      P[g] = P[g] + 1 ;
6  query value () : integer v
7      let v = ∑_i P[i] ;
8  compare (X,Y) : boolean b
9      let b = (∀i ∈ [0,n−1]) : X[i] ≤ Y[i]) ;
10 merge (X,Y) : payload Z
11     let ∀i ∈ [0,n−1] : Z[i] = max(X[i],Y[i]);
```

The payload is a vector of integers, where each entry represents a replica. When a replica wants to increment adds 1 to the respective entry. To check the value of the counter, it is just needed to sum all the entries of the vector. The comparison of states is made with the partial order: $X \leq Y$, if all of X elements are lower or equal to the respective elements in Y. Merge operates taking the maximum of each entry. This data type is a CvRDT because the merge operation produces the LUB and his states form a *monotonic join semilattice* since the payload after *increment* respects the partial order.

In this example were made two assumptions that should be mention. The integer payload does not overflow, and contrary to what is expected in state-based, is assumed to know the full set of replicas.

Sets

Sets are data structures where there are no repeated elements. The sequential specification of a set has two fundamental operations: add, performing the union of a new element with the set; and remove, which performs the set minus with the element. These operations are

not commutative, factor that complicates the CRDT desgin. For an element x of a given set: $add(x).remove(x) \neq remove(x).add(x)$.

G-SET The Grow-only Set is the simplest of CRDT sets, due to the fact that the operation *remove* is not allowed, supporting only *add* and *lookup*. Specification 2 shows a state-based approach for the G-Set.

Specification 2: State-based grow only-set

```
1  payload set A
2      initial ∅ ;
3  update add (element e)
4      A := A ∪ {e} ;
5  query lookup ( element e) : boolean b
6      let b = (e ∈ A) ;
7  compare (S,T) : boolean b
8      let b = S ⊆ T ;
9  merge (S,T) : payload U
10     let U = S ∪ T;
```

As expected, this approach works even when the number of replicas is not known. The partial order is given by set inclusion. The operation *add* respects the partial order because the set only grows and *merge* produces the LUB. So, we are in the presence of an CvRDT.

2.4.3 *Commutative Replicated Data Types (Operation-based)*

In operation-based, replication is active, and an update is sent to all replicas in the form of an operation, requiring to know the full set of replicas. A *reliable broadcast* is needed, in order to guarantee that every replica receives an update. Updates need to be in a *delivery order* $<_d$, specified by the data type. An operation not ordered by the delivery order $<_d$ is said *concurrent*.

Concurrent operations: *two operations f and g are concurrent ($f||_d g$) iff: $f \not<_d g$ and $g \not<_d f$.*

The delivery order needs to ensure that all concurrent operations commute in order to all replicas converge to the same state. Operation-based CRDTs use *causal delivery* $<_{\rightarrow}$ which satisfies the delivery order.

Causal history (Operation-based): *The causal history of a replica x_i ($\mathcal{C}(x_i)$) is defined as follows:*

- *Initially,* $\mathcal{C}(x_i) = \emptyset$.
- *After executing the downstream phase of* f *at replica* x_i: $\mathcal{C}(f(x_i)) = \mathcal{C}(x_i) \cup \{f\}$.

Happens-before (→): $f \rightarrow g \Leftrightarrow \mathcal{C}(f) \subset \mathcal{C}(g)$.

Causal delivery: *if* $f \rightarrow g$ *then* f *is delivered before* g.

Updates need to be delivered *exactly once*, which means that we can't apply an update more than one time and we can't miss updates, but the last is assured by a *reliable broadcast*.

2.4.4 *Summary*

CRDTs operate under the CAP constraints, using a weak form of consistency to provide high-availability. There are two kinds of CRDTs: CvRDTs and CmRDTs.

CvRDTs send updates as state, using passive replication, and are simpler to reason about since the state captures all necessary information. Also, they require weak channel assumptions allowing any number of replicas. However, sending the entire state could be very inefficient and, sometimes, simple sequential specifications require some effort to design a CvRDT. CmRDTs send updates as operations, using active replication and are dependent on the communication channel implementation. Also, they are more complex since they require reasoning about history. However, they have greater expressive power and, sometimes, the payload could be simpler.

2.5 MERGEABLE OBJECTS

CRDTs are not designed to perform in shared memory. In large-scale distributed systems, the merge cost is negligible when compared to network-based synchronization, but for our purpose it is inefficient. In order for the merge operation to be efficient, we will need *mergeable objects* properties. First introduced by *Deepthi Devaki Akkoorath and Annette Bieniusa* in *Transactions on Mergeable Objects* [3], and later adapted in *Highly-scalable concurrent objects* [4].

A mutable shared object O is said to be *mergeable* when concurrent updates can be merged meaningfully. Two main properties of these objects were identified: *persistence* and *mergeability*.

A data structure is said to be *persistent* if multiple versions of it are accessible, i.e, older versions of the data structure are preserved when in the presence of modifications. This is effectively related to immutability since updates create newer versions. This is an important property because it allows multiple versions of the same object, and, consequently, concurrent accesses to multiple-versions. *Persistence* could be *partial* if all versions are ac-

cessible but, only some versions of it are modifiable, or *full* if all versions are accessible and modifiable. If a data structure is fully persistent and there is also a merge operation that creates a new version from two versions, the data structure is said to be *confluently persistent*. The ability to merge branches is what differentiates these two types of persistence. Persistence could be achieved by different mechanisms regarding the implementation and it is important to know which versions must persist.

Mergeability is the ability to merge different versions of the same object in a correct new version. Two different aspects are worth mention in this field: *semantic mergeability* and *structural mergeability*. Semantic is related to the merge operation and how data types behave after the merge. Structural is associated with the efficiency of the merge and with low-level implementation details.

In section 3.1 we will see how these properties relate directly to MDTs.

3

MERGEABLE DATA TYPES(MDTS)

The concept of *Mergeable Data Types* (MDTs) appeared on *Highly-scalable concurrent objects* by *Deepthi Devaki Akkoorath and Annette Bieniusa* [4]. Motivated by the fact that, currently, in shared memory environments, synchronization is needed and is a bottleneck. MDTs, are convergent data structures, used to relax synchronization in shared memory environments, analogously to CRDTs in distributed systems.

Data structures that implement an abstract data type are often called objects and, MDTs central idea is that *each thread can update its local private copy of any object and later merge its changes to a shared global object*.

Later, in *Global-Local View: Scalable Consistency for Concurrent Data Types* [5], we have proposed a new programming model for concurrent access to objects in a shared memory system, defining a specification for the model and providing the respective correctness definitions.

3.1 PERSISTENCE AND MERGEABILITY

Following the work on *Highly-scalable concurrent objects* [4], mentioned on Section 2.5, we can see an MDT as a mergeable object O. An object that provides a global copy with the last committed version, which is accessible to all threads. Also, an interface with methods for *merge* and *updates* is provided. When a thread wants to update O, makes a thread-local copy of the object and uses the available update methods on the interface to change his copy. Then, the changes to the global copy are committed using the merge function.

Thread-local and global versions co-exist, thread-local versions can be modified by the respective thread and the global version can be modified by merging thread-local operations, ensuring that MDTs are *confluently persistent* data structures. Different versions should persist while threads are doing effective work. After that, non-used versions should be garbage collected.

The behavior of the merge operation should be deterministic to guarantee that any concurrent updates can merge their results and obtain a consistent deterministic state. Also, the updates should be reflected in the merged version (*semantic mergeability*).

MDTs should have a well-defined semantics of concurrent updates and the semantics of merging two different versions. Every thread-local instance must be mergeable with the global shared object, but local versions from different threads do not need to be merged (*structural mergeability*).

3.2 GLOBAL-LOCAL VIEW PROGRAMMING MODEL

As the name suggests, each thread maintains different views on the shared object. A *local view*, or *thread-local view*, where threads can read and perform updates without synchronization costs because there is no shared data. And, a *global view*, or *shared global view*, that is visible to every thread.

Local updates become visible to other threads only after the thread-local view is merged with the shared global view. And, threads observe the changes once they synchronize, by merge, their local view with the global view. This is a result of the merge being an explicit *two-way merge operation*.

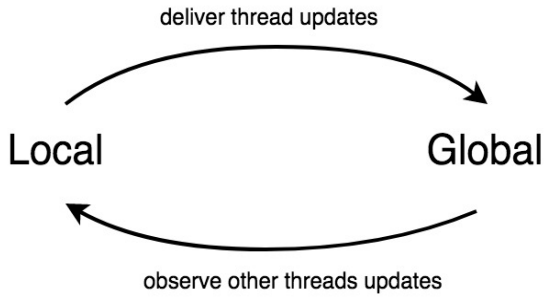


Figure 9: Two-way merge operation

Moreover, if all operations in the data type are commutative, there will be no conflicts at merge time. If we have non-commutative operations, the conflicts are resolved by a type-specific merge operation.

In addition to local operations, the model also provides synchronous operations on the global view. Operations that perform only on the local view are called *weak operations*, and those on the global view, *strong operations*. Combining operations on the global and local views, give us the flexibility to build data types with customizable semantics on the spectrum between sequential and purely mergeable data types.

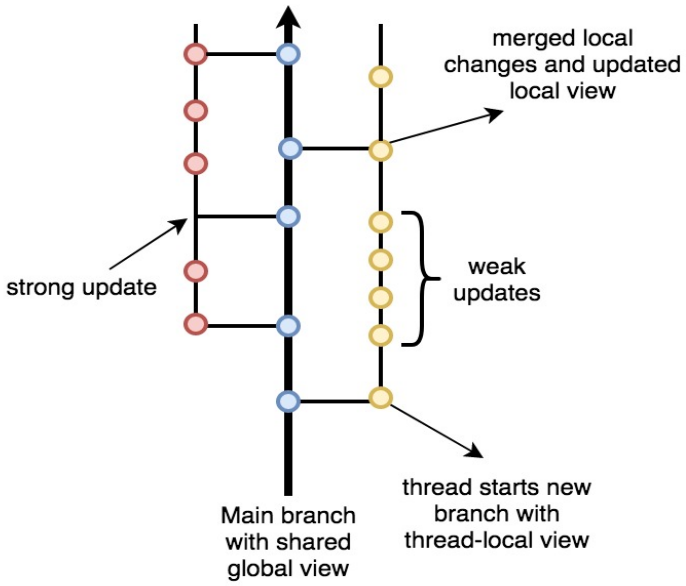


Figure 10: Mergeable Data Types basic workflow

We will designate data types that only provide weak operations as *Mergeable Data Types* and the ones that offer both weak and strong operations as *Hybrid Mergeable Data Types(HMDTs)*. An application that uses HMDTs may use weak updates when non-linearizable access is sufficient and can switch to use only strong operations when stronger guarantees are required. A good example is a queue where enqueues are executed on the local view and, to ensure that elements are dequeued only once, dequeues are executed atomically on the global view. Better performance at the expense of linearizability because threads can do work without synchronization costs and we have simple correctness criteria to reason about.

3.3 System model

Consider a system built upon the classic shared memory-architecture as supported by the memory models at C++ and Java languages. Where we have a variable number of threads and any thread can spawn new threads that may outlive their parents. Also, threads do not fail independently.

A system where we can distinguish different memory types. Specifically, a thread-local memory that can be accessed only by the respective thread and a shared memory that can be accessed by anyone.

The communication between threads is done via shared memory objects where threads and objects are identified by a unique identifier. Also, threads do not inherit any local objects from their parents.

We assume an abstract global time that can be used to determine relative ordering of events happening in concurrent threads.

3.4 NOTATION

An operation opKind on an object can be formalized as a function:

$$\mathsf{opKind}_t(m, g, s_t, l_t) = (r, g', s'_t, l'_t)$$

The first parameter m represents an optional type-specific update(u) or query(q) method applied on the object, g denotes the shared global object on which the operation is applied, and t is a thread identifier that refers to the non-shared local version (s_t, l_t) of the object. Where, s_t denotes a local snapshot of the shared object state g which gets updated upon synchronization, and l_t refers to the local updates not yet incorporated in the shared global state g. The operation returns a tuple (r, g', s'_t, l'_t) where r is the return value of the method m and the other variables refer to the updated global g' and local state s'_t,l'_t. State variables – g, s_t, l_t – are each modeled as a sequence of updates, initially empty; a sequence x can be concatenated with another sequence y (or a single update), denoted by $x \cdot y$.

3.5 SPECIFICATION

Using the previous notation, basic operations that are type-independent can be specified as follows:

- **pull** updates the local object snapshot with the global object state; local operations are not changed.
$$\mathsf{pull}_t(g, s_t, l_t) = (\bot, g, g, l_t)$$
- **weakRead** returns the result of a type-specific read-only operation q on the state obtained by applying local updates on the local snapshot.
$$\mathsf{weakRead}_t(q, g, s_t, l_t) = (q(s_t \cdot l_t), g, s_t, l_t)$$
- **strongRead** returns the result of a type-specific read-only operation q on the state obtained by applying local updates on global state. Neither the global state or the local snapshot are changed.
$$\mathsf{strongRead}_t(q, g, s_t, l_t) = (q(g \cdot l_t), g, s_t, l_t)$$

- **weakUpdate** applies the update method u on the local copy without any synchronization to the global state.

$$\mathsf{weakUpdate}_t(u, g, s_t, l_t) = (s_t \cdot l_t \cdot u, g, s_t, l_t \cdot u)$$

- **strongUpdate** applies the update method u on the global state atomically. The previous weak updates that are batched in l_t are not merged at this point.

$$\mathsf{strongUpdate}_t(u, g, s_t, l_t) = (g \cdot u, g \cdot u, s_t, l_t)$$

- **merge** incorporates the local updates to the global state and updates the local snapshot.

$$\mathsf{merge}_t(g, s_t, l_t) = (\bot, g', g', \bot)$$

  where $g' = merge(g, (s_t, l_t))$ and *merge* is type specific merge operation. In general, if the updates are commutative, $g' = g \cdot l_t$. The data types can also specify a conflict resolving *merge* operation, in case of non-commutative concurrent updates.

While weakRead and weakUpdate act exclusively on the local copy, strongRead and strongUpdate act on the global state. The combination of these two operations supports flexible optimizations on each individual data type.

3.6 Relation with CRDTs

MDTs fit its own corner in the design space, but we can see some resemblances with CRDTs, either state-based or operation-based. In a general way, the shared-memory environment leverages us to a simpler approach, since we have fewer factors to consider. For instance, we don't need to concern about messages lost due to network errors, which means that we can immediately update the thread-local/global state.

Synchronization is avoided akin to CRDTs, allowing threads to do effective work on its own version while converging to the same global version trough merge. The merge operation is idempotent like in the state-based CDRTs. Every time the merge is performed the local pending updates l'_t are incorporated, so performing the merge without local pending updates, will, as much, only update the local snapshot s'_t.

A way of performing updates is by merging state as in state-based CRDTs. Moreover, if we have commutative operations, the order of merge does not influence the result. However, the applicability of CRDTs as described in literature [33] is limited in a concurrent shared memory environment. For example, a CRDT counter is implemented as a map of replica id to an integer. The merge operation iterates over the two maps to be merged and returns a map with the maximum for each entry. Thus, the relative cost in space and time of the

merge is linear in the number of entries and as such unfeasibly high. In the global-local view model, the merge is executed synchronously on the global view. If the cost of merge is high, we lose the benefits of allowing parallel updates.

Results of queries can be determined by the partial order of updates that are known (by visibility) at any given point. Merge is just a way of making operations that were only locally known in a thread, visible to other threads. On the other hand, contrasts with state-based, where all versions from different replicas should be *mergeable*. Two thread-local versions don't need to be *mergeable* since every thread-local versions are branched from the global version and committed back to it.

On thread identity, we don't need to know the set of threads involved, since they can initialize their own state. Although knowing them is useful for joins, it does not look to be a mandatory property as is the knowledge of the group of replicas in operation-based, for the broadcast operation.

3.7 SUMMARY

To our knowledge, the GL view programming model introduces the first consolidated theory on mergeable data types in the shared memory ecosystem by proposing a specification that allows building any data type.

Fewer restrictions to concurrency are presented since we only need synchronization when the merge operation is performed or when in the presence of strong operations.

CRDTs introduced the fundamentals, and we can establish many resemblances between the data types. Persistence and mergeability give us important properties to achieve a suitable merge operation for the environment.

In the next section, we present our portfolio of mergeable data types, tailored for shared memory concurrent programs.

4

PORTFOLIO OF MDTS

Each mergeable type defines a subset of the basic operations from the global-local view model, depending on the semantics needed. A purely mergeable counter defines only weak operations and merge, while a hybrid mergeable counter also defines strong operations. In this section, we discuss the specification of several data types and a design pattern for their implementation.

4.1 SPECIFICATION

4.1.1 *P-Counter*

As we have seen on Section 2.4.2, a counter is a replicated integer supporting the update operation of *increment*, which increases the counter by one unit, and the operation *value* for reads.

In a shared counter, the final value should reflect all the increments, but the order between them is not relevant since they are commutative. If each increment by each thread is an atomic operation made visible to all other threads, it can become a bottleneck. This situation leverages an approach such as the one presented in our programming model because it is sufficient to execute the increment on a thread-local variable and to apply a combined update to the shared object.

The operations of a shared counter are presented by the specification as follows:

- $\mathsf{weakValue}_t() = \mathsf{weakRead}_t(\mathit{value}, \_, s_t, l_t)$
- $\mathsf{weakInc}_t() = \mathsf{weakUpdate}_t(\mathit{inc}, \_, \_, l_t)$
- $\mathsf{merge}(g, (s_t, l_t)) = g \cdot l_t$
- $\mathsf{strongInc}_t() = \mathsf{strongUpdate}_t(\mathit{inc}, g, \_, \_)$
- $\mathsf{strongValue}_t() = \mathsf{strongRead}_t(\mathit{value}, g, \_, l_t)$

A purely mergeable counter could be given by the first three operations. weakValue returns the value of the counter considering only the local snapshot and the not incorporated local operations; weakInc increments only locally; merge appends the local increments to the global sequence g, because the increments are commutative.

A hybrid mergeable counter could be provided with the addition of strongInc, that increments synchronously on the global copy and strongRead that returns the value of the counter considering the not incorporated yet local increments and the global shared copy.

Without much effort we can have a *PN-Counter* by adding a *decrement* operation that could be specified similarly to the previous update operations:

- $\mathsf{weakDec}_t() = \mathsf{weakUpdate}_t(dec, \_, \_, l_t)$
- $\mathsf{strongDec}_t() = \mathsf{strongUpdate}_t(dec, g, \_, \_)$

4.1.2 *Add-only Bag*

A bag, or multiset, is an unordered collection of elements with duplicates. An add-only bag has available the *add* operation for updates and the *lookup* operation for reads. The update operation adds one element to the bag, and the lookup operation verifies if a given element is present.

Since the order at which elements are added does not affect the final state of the bag, this is another good situation to apply the global-local view model.

The operations are presented by the specification as follows:

- $\mathsf{weakLookup}_t(e) = \mathsf{weakRead}_t(lookup(e), \_, s_t, l_t)$
- $\mathsf{weakAdd}_t(e) = \mathsf{weakUpdate}_t(add(e), \_, \_, l_t)$
- $\mathsf{merge}(g, (s_t, l_t)) = g \cdot l_t$
- $\mathsf{strongAdd}_t(e) = \mathsf{strongUpdate}_t(add(e), g, \_, \_)$
- $\mathsf{strongLookup}_t(e) = \mathsf{strongRead}_t(lookup(e), g, \_, l_t)$

weakLookup will search for a given element in the local pending operations and local snapshot returning if the element is present or not; weakAdd will add the element to the local pending operations; merge will simply add the local pending elements to the global bag because the update operation add is commutative. With these three operations, we can have a purely mergeable add-only bag, allowing threads to concurrently and efficiently add elements without violating its semantical correction.

If more restricted guarantees are needed, we can switch to stronger operations. A hybrid mergeable data type can be specified by adding strongAdd, where elements are added

directly to the global bag, and strongLookup, where the lookup is performed on the global copy and in the local not incorporated operations. It is preferable in terms of efficiency if first, the lookup starts on the local pending operations and only after moves to the shared global copy allowing elements on the bag to be found without requiring any synchronization.

This is a case were might be inefficient/unnecessary to store a full snapshot of the bag's global view on the local view. In Section 5 we show how we can benefit from the data type semantics to mitigate this problem.

4.1.3 *OR-Set*

An *Observed-Removed Set(OR-Set)* is a collection without duplicated elements which supports the update operations of *add* and *remove*. These two operations, under the same element, are not commutative between them and might lead to conflicts: $\mathsf{add(e)} \cdot \mathsf{remove(e)} \neq \mathsf{remove(e)} \cdot \mathsf{add(e)}$.

Concurrent adds of the same element make the element available in the set. On merge, the two concurrently added elements are considered as the same element. When two versions merge, where one has an add operation and the other a concurrent remove of the same element, then the resulting version retains the element in the set(*add-wins*). An element can only be removed if the thread that is trying to remove it can observe the element (*observed-removed*). A *lookup* is also provided for reads, where given an element returns if the element is present in the set or not.

The operations are presented by the specification as follows:

- $\mathsf{weakLookup}_t(e) = \mathsf{weakRead}_t(lookup(e), \_, s_t, l_t)$
- $\mathsf{weakAdd}_t(e) = \mathsf{weakUpdate}_t(add(e), \_, \_, l_t)$
- $\mathsf{weakRemove}_t(e) = \mathsf{weakUpdate}_t(remove(e), \_, \_, l_t)$
- $\mathsf{merge}(g, (s_t, l_t)) = g \cdot l_t$
- $\mathsf{strongAdd}_t(e) = \mathsf{strongUpdate}_t(add(e), g, \_, \_)$
- $\mathsf{strongRemove}_t(e) = \mathsf{strongUpdate}_t(remove(e), g, \_, \_)$
- $\mathsf{strongLookup}_t() = \mathsf{strongRead}_t(lookup(e), g, \_, l_t)$

weakAdd, strongAdd, weakLookup and strongLookup will behave as in the Add-only Bag. Since elements can only be removed if observed by a given thread, weakRemove will have to check in the local pending operations if there is a local add for the same element, and in the local snapshot if the element can be observed; strongRemove removes the element if

present on the global copy; merge performs the local pending updates on the global copy, having in consideration the *add-wins* semantics.

The detection of concurrent adds and removes is delegated to the implementation. In Section 5 we show how we use version numbers to do it.

4.1.4 *Queues*

A sequential queue is a collection of elements that are inserted and removed according to the first-in first-out (FIFO) principle. The update operations $enqueue(e)$ and $dequeue$ are provided. A hybrid mergeable queue with mergeable enqueue and synchronized dequeue defines the following operations:

- $\mathsf{enqueue}_t(e) = \mathsf{weakUpdate}_t(enqueue(e), \_, \_, l_t)$
- $\mathsf{dequeue}_t() = \mathsf{strongUpdate}_t(dequeue, g, \_, \_)$
- $\mathsf{merge}(g, (s_t, l_t)) = gl_t$

In the above semantics, if the global copy is empty, *dequeue* returns null even if there are local enqueue operations by the same thread which have not been merged yet. We can allow dequeue to include local enqueue operations by defining:

$$\mathsf{dequeue}_t() = \mathsf{strongUpdate}_t(dequeue, g', \_, \_) \text{ with } (\_, g', \_, \_) = \mathsf{merge}_t(g, s_t, l_t).$$

This way we can combine operations to give different semantics. For example, a queue with weak enqueue and weak dequeue may be useful if redundant dequeue is not a problem for the application. A queue with both strong enqueue and strong dequeue behaves as a linearizable queue.

4.2 IMPLEMENTATION PATTERN

Mergeable data types consist of two parts - a local view and global view that may or may not be of the same type. From the specification, a generic design pattern for implementations could be described by the following object-oriented programming pseudo-code:

```
1 type MDT {
2   ThreadLocal  T1 localView;
3   T2 globalView;
4
5   weakRead(param){
6     localView.read(param);
7   }
```

```
8
9   weakUpdate(param){
10    localView.update(param);
11  }
12
13  strongRead(param){
14    atomic {
15       globalView.read(param, localView);
16    }
17  }
18
19  strongUpdate(param){
20    atomic {
21      globalView.update(param);
22    }
23  }
24
25  merge(){
26    atomic {
27       globalView.merge(localView);
28       localView.reset(globalView);
29    }
30  }
```

The types of localView and globalView (**T1**,**T2**) may or may not be the same. **localView** comprises the local pending operations and the local snapshot and are understood as thread-local instances as identified by **ThreadLocal**. A variable specified as **ThreadLocal** exists per thread in the thread's private storage and guarantees that each thread is accessing its own private view. Many programming languages support some form of thread-local storage (TLS). A mergeable data type can also implement its thread local storage by mapping thread ids to different instances of the object.

atomic refers to any synchronization mechanism, such as mutex or compare and swap, that atomically executes the code block within.

Regarding the operations, the **param** argument in each operation is optional. **weak** operations act only on the localView. **strongUpdate** performs updates synchronously on the globalview. **strongRead** returns a query on the globalView that considers information on the localView, i.e., the non-incorporated updates. **merge** synchronously incorporates the updates on the localView into the global view. **reset** updates the localView accordingly with globalView.

4.3 SUMMARY

The GL view model can be used to specify any mergeable data type. We have provided a portfolio with the specification of some known data types, leveraging efficiency by maintaining and using the data types semantics.

We can also use the model to provide different semantics on the same data types as in the *queue* specification.

The model specification translates to a general design pattern that will be used in our implementations of the data types in the next chapter.

5

EVALUATION

In this chapter we will evaluate some of the theoretical contributions made among this dissertation, answering the questions:

- Is a thread local approach, such as the one presented, efficient when compared with the existing concurrency mechanisms?
- Are CRDTs inefficient for shared memory environments?
- Are MDTs more efficient than the respective linearizable versions of the same data type?
- Are MDTs efficient in situations where no divergence is allowed between threads?
- Do MDTs have real-world scenarios to be applied?

All the code used to implement and benchmark the MDTs is written in C++ and can be found at `https://github.com/josebrandao13/MDTs`.

The evaluations were performed on a 12 core CPU (2 NUMA nodes) with 2-way hyper-threading on a Ubuntu 16.04 OS. The compiler used was CLANG++ with C++14 support. We also used the widely known Boost C++ Libraries in the version 1.58 .

5.1 THREAD-LOCAL STORAGE

Thread-Local Storage (TLS) is a dedicated storage area that can only be accessed by one thread. Specifically, each thread accesses an independently initialized copy of the state that is only available at the thread's scope and where changes only persist during thread's execution.

5.1.1 *Proof of Concept*

In this section, we will compare a thread-local approach to handle concurrency with some of the existing mechanisms on the C++ language. To do it, we have implemented four versions of a simple counter that stops after each thread performs a given number of increments.

A first version of the counter with *no synchronization* mechanism that we will refer as *no sync* counter; a version implemented using the traditional mutual exclusion locks (*lock counter*); another version using C++ atomic types (*atomic counter*); and a *thread-local counter*, implemented using Boost's thread specific ptr.

```
1  unsigned int c = 0;
2  atomic<unsigned int> v;
3  mutex m;
4  boost::thread_specific_ptr<unsigned int> t;
5
6  void noSyncInc(int incNumber){
7    for (int i = 0;  i < incNumber;  i++){ c++; }
8  }
9
10 void lockInc(int incNumber){
11   for (int i = 0;  i < incNumber;  i++){
12     m.lock();
13     c++;
14     m.unlock();
15   }
16 }
17
18 void atomicInc(int incNumber){
19   for (int i = 0;  i < incNumber;  i++){ v++; }
20 }
21
22 void threadLocalInc(int incNumber){
23   t.reset(new unsigned int(0)); //init thread-local var
24   for (int i = 0;  i < incNumber;  i++){
25     *t += 1;
26   }
27   m.lock();
28   c = c + *t; //add thread-local increments to a shared var
29   m.unlock();
30 }
```

As we can see from the previous code snippet, in the *thread-local counter* each thread increments a thread-local variable, and at the end, adds its increments synchronously to a shared variable. In the *atomic counter* we don't need to concern about concurrency due to the fact that we are using C++ atomic types. It makes use of lock-free mechanisms in the implementation of some operators, if multiple threads try to increment the counter the behavior is well-defined, and consequently, we have no data races. It is a more straightforward and performant approach when compared to mutex.

To compare the four counters we have done two benchmarks: a first one where we have a fixed number of increments per thread (1 million) and will gradually increase the number of threads; and a second one where for a fixed number of threads(24=2*number of machine cores) we had increased the number of increments to be done. The results were the following:

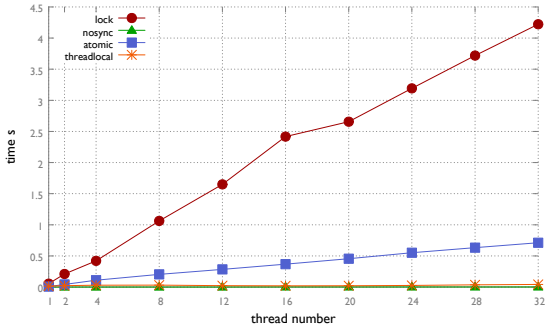


Figure 11: Time to perform 1 million increments increasing the number of threads

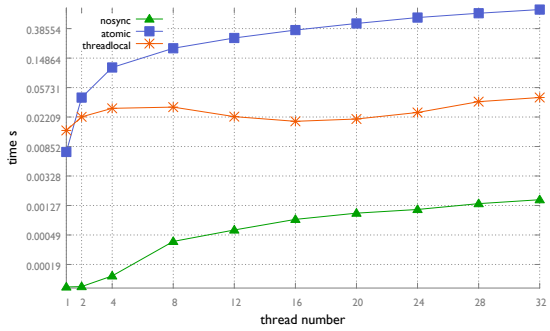


Figure 12: Time to perform 1 million increments increasing the number of threads (excluding lock counter)

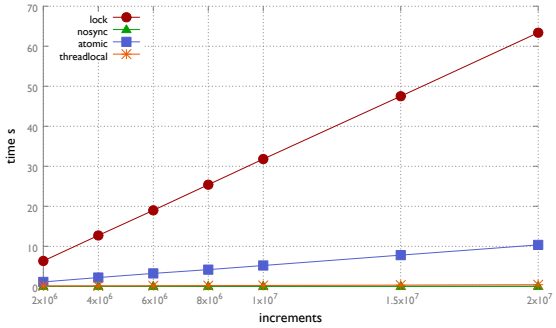


Figure 13: Time to perform a variable number of increments with 24 threads

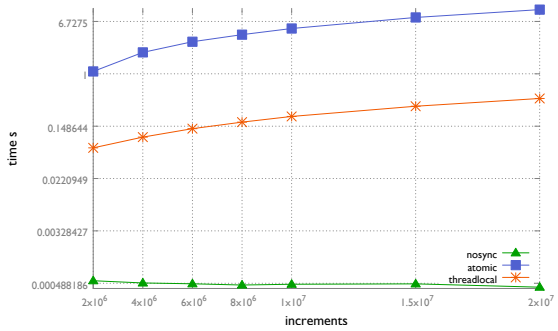


Figure 14: Time to perform a variable number of increments with 24 threads (excluding lock counter)

The *lock counter* was by far in both benchmarks the one whose performance was worst, where the values were completely out of bounds relatively to the other versions.

In opposition and as also expected, the *no sync* counter achieved the best results but mostly failed on achieving the target value of the counter due to concurrency not being handled.

With the *atomic* counter, we were able to correctly process from 2M operations to 20M in a time interval of 1 to 10 seconds at the machine's optimal number of threads(24). When increasing the number of threads for the same number of increments (1M) the processing time has not decreased.

Finally, the *thread-local* counter was able to correctly process the same number of operations in a time interval between 0.06 and 0.4 seconds. For a fixed number of increments, as we have increased the number of threads, we could see that the time needed to process them was reducing until we reach the machine's optimal thread number of 24.

These results gave us the motivation to keep exploring the thread-local approach in other scenarios.

5.1.2 *Why use Boost libraries ?*

The main motivation under the use of *Boost libraries* was their flexible and performant thread-local storage implementation. We have excluded C++ standard thread-local implementation merely because of the limitations imposed to create a clean object-oriented design. The *thread_local* identifier is not allowed on variables without the *static* identifier. In order to have *static* identifiers inside of instance variables it is required for those variables to be initialized outside of classes, not allowing to encapsulate all the logic needed inside the class definition.

Boost, on the other side, allows the use of thread_specific_ptr inside of a class definition without these concerns. The *thread_specific_ptr* represents a pointer to an object where each thread must have a distinct value, providing a thread-local storage mechanism.

An alternative is a *manual thread-local* strategy, implemented by us, where the idea consists on a map of thread ids to objects, where each thread only changes the respective entry.

We have compared this last manual mechanism with Boost's in order to choose the one that performed better. We have used a purely mergeable P-Counter class to emulate the complete experience of implementing a mergeable data type. The code can be found here.

The benchmark consists on making threads increment the counter until reaching a value of 50 million. Throughput is measured by changing the number of threads and the synchronization intervals (increments). The results are the average between 5 runs for the same number of threads and synchronization frequency (as in every benchmark results presented on this dissertation).

The following plot presents the results for threads synchronizing in intervals of 64, 512 and 4096 increments:

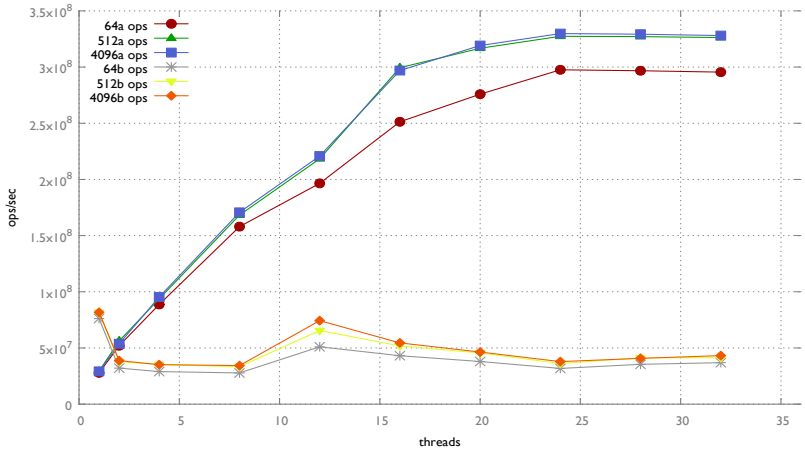


Figure 15: Manual thread-local(a) vs Boost thread-local(b)

The manual version is clearly outperformed by the Boost's one. In fact, it scales poorly when increasing the number of threads or synchronization frequency.

As expected in a purely mergeable counter, the two implementations overshoot the target value and for a matter of simplification and performance perspective we are not presenting overshoot information but we will do it in detail on P-Counter evaluation at Section 5.2.

Our technology choice ended up in Boost due to good performance and the fact that we wanted MDTs to have a clean design, to be programmer-friendly and to deliver a single object that encapsulates all the needed logic. This way the fact that MDTs deal with thread-local storage and concurrency mechanisms is not a concern to the final user.

Boost libraries also provided other useful libraries for this dissertation: Lock free data structures, Chrono and Synchronization mechanisms.

5.2 P-COUNTER

The P-Counter was the example used to decide on which of the different technology options should we go. This chapter will focus on making a deeper benchmark, starting by comparing a purely mergeable P-Counter with an atomic counter using only the compare and swap operation and later with a hybrid mergeable P-counter.

The following code snippet shows our implementation of the P-Counter:

```
1 template <typename V = int> class counter {
2 private:
3   atomic<V> gcount;
4   mutex m;
5   boost::thread_specific_ptr<V> lcount;
6   boost::thread_specific_ptr<V> lgcount;
```

```
7
8 public:
9
10  void init(){ //instatiate thread local object parts
11     lcount.reset(new V(0));
12     lgcount.reset(new V(0));
13   }
14
15   void weakInc(V tosum = {1}){ //weakUpdate
16     *lcount += tosum;
17   }
18
19   V weakValue(){ //weakRead
20     V readed = *lgcount + *lcount;
21     return readed;
22   }
23
24   void merge(){ //mergeToGlobal
25     gcount.fetch_add(*lcount); //merging thread local state with global state
26     *lgcount = gcount;  // updating the snapshot
27     *lcount = 0; //resetting thread local state
28   }
```

The *init* operation will be used to instantiate the thread local variables, in this case, the per-thread counter and the per-thread counter snapshot (*lgcounter*).

The *fetch_add* operation atomically replaces the current value with the result of adding the new value. The operation is read-modify-write, and the memory order used by default is *std* :: *memory_order_seq_cst*, meaning that it respects the correctness condition of sequential consistency. A read-modify-write operation in this memory order performs both an acquire operation and a release operation, plus a single total order exists in which all threads observe all modifications in the same order. In our P-Counter the operation is used at merge time to aggregate the values of the different threads on the global counter.

We are using c++ templates in order to provide more generic implementations, allowing the counter to be extended to other c++ data types that accept the available operations.

The atomic counter that we built separately makes use of a compare and swap operation:

```
1 atomic<unsigned int> c;
2
3 int inc(){
4     unsigned int oldValue = c.load();
5     unsigned int newValue;
6
7     do{
8         if (oldValue < TARGET) newValue = oldValue + 1;
```

```
9           else break;
10      }
11      while (!c.compare_exchange_weak(oldValue, newValue));
12
13      return oldValue;
14  }
```

The increment operation starts by atomically loading the value of the counter to a variable. Then it will make use of the *compare_exchange_weak* operation to try to change the value of the counter atomically. In more detail, the operation starts by comparing the *oldValue* with the counter value, if they are the same then it will update the counter value with the *newValue*, that is equal to the previous one plus one unit, and return true making the cycle to finish and consequently the increment operation. If is not equal, it means that another thread has changed it and the compare_exchange_weak operation will return false and it will update the *oldValue* with the latest counter value, iterating until the compare_exchange_weak returns true. When the *oldValue* is already equal to the target value, the inc operation simply returns the value.

In the first experiment, we allow threads to increment the shared purely mergeable P-Counter until a target value of 50 million is reached and compared it with the atomic counter. Since threads might not know about non-merged increments from other threads, they typically end up overshooting the target on the purely mergeable version. As in the previous section, we evaluated several merge-intervals, labeled with how many local increments are allowed between merges. The full code can be found here.

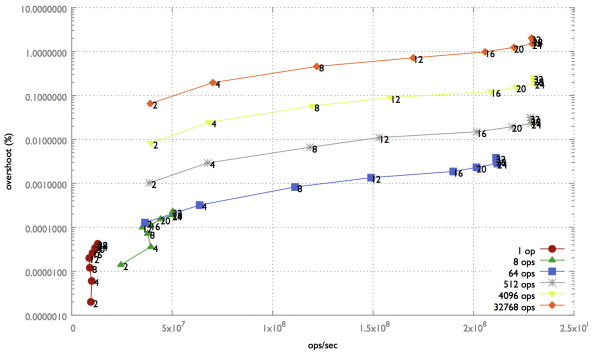


Figure 16: Purely mergeable P-counter overshoot

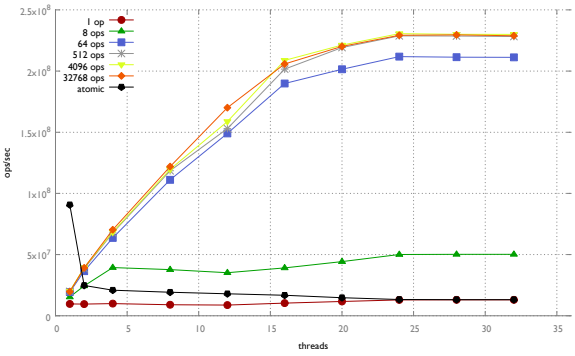


Figure 17: P-counter vs atomic counter

Figure 16 shows that the throughput scales linearly with the number of threads and with the merge-intervals. At the same time, the overshoot increases. However, the percentage of the overshoot is small. (Notice that overshoot is upper bound by the number of threads times the merge-interval, as this reflects the number of increments not yet accounted for).

The atomic version in Figure 17, never overshoots the target, but since threads are always competing on the increment, performance is very low, and no speedup is obtained. In contrast, the mergeable counter can scale linearly up to a good fraction of the available concurrency, in particular with merge-interval of ≥ 4096.

While some applications could tolerate an overshoot, in general, applications will require to further bound the overshoot. To address this, we provide a variant of the mergeable counter that makes a hybrid use of initial weak local increments and later switches to strong atomic increments (with our atomic counter) when approaching the target. The first thread that, upon the periodic merges, detects that it is close to the target, initiates a barrier synchronization to ensure that all threads have switched to strong operations.

The entire code used to perform the barrier and the hybrid P-counter can be found here. A snippet of the code is presented:

```
bool switchToStrong = false;
int THRESHOLD = TARGET - ( THREAD_NUMBER * SYNQ_FREQ );
int thresholdThreadNumber = 0;
boost::condition_variable cond;
boost::mutex mut;

for (int i = 0;  i < LOOP;  i++){
    if(!switchToStrong){
        if(mdt.weakValue() >= THRESHOLD) {
            mut.lock();
            thresholdThreadNumber++;

            if(thresholdThreadNumber == THREAD_NUMBER) {
              cond.notify_all();
            }
            mut.unlock();

            boost::unique_lock<boost::mutex> lock(mut);
            while(thresholdThreadNumber < THREAD_NUMBER){
                cond.wait(lock);
            }

            switchToStrong = true;
            continue;
        }
        performWeakInc();
    }

    else{
        if (mdt.atomicInc() >= TARGET){
            break;
```

```
32          }
33      }
34  }
```

In this case, the chosen threshold was a relationship that we have established between the thread number and the synchronization frequency regarding the target value. Threads will perform weak increments while *swicthToStrong* is false. As soon a thread weakly detects that has passed the defined threshold (line 25), we increment *thresholdThreadNumber* and use boost::condition_variables to make the thread wait on a condition. The last thread that detects that the threshold was surpassed will notify all threads that are waiting. After that, *swicthToStrong* is updated, and only strong atomic increments are performed until the target is achieved without overshoot.

In the second experiment, we have repeated the previous scenario of threads incrementing a counter until a target value of 50 million is reached, but this time using the hybrid P-Counter version:

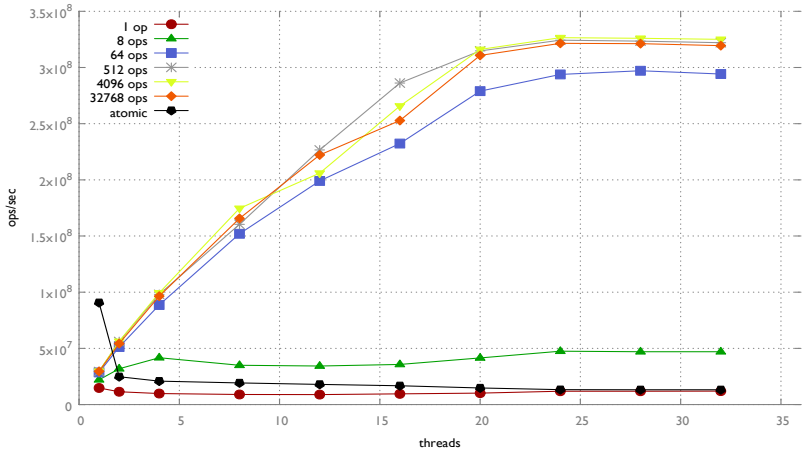


Figure 18: Hybrid P-Counter vs atomic counter

Figure 18 shows that under this approach, overshoot is eliminated while the performance is almost identical to the mergeable counter.

A PN-Counter could be easily implemented by adding weak and strong decrement operations to the P-Counter:

```
1
2 void weakDec(V tosum = {1}){
3     *lcount -= tosum;
4 }
5
6 void strongDec(V tosub = {1}){
```

```
7       gcount.fetch_sub(tosub);
8   }
```

5.2.1 *P-Counter CRDT evaluation*

In this experiment, we demonstrate that CRDT's design has significant overhead when used in a shared memory environment. We have implemented a P-Counter CRDT on the global-local view model, where each local view and global view are a CRDT replica. The counter value is obtained by getting all values on a replica, and the operation merge is the maximum of all values on a local replica and the global one.

We have built two versions, a first one where a replica is represented using a Map (C++ standard map) of thread-id to counter and a second one using an Array where each index corresponds to a thread-id. The code can be found here.

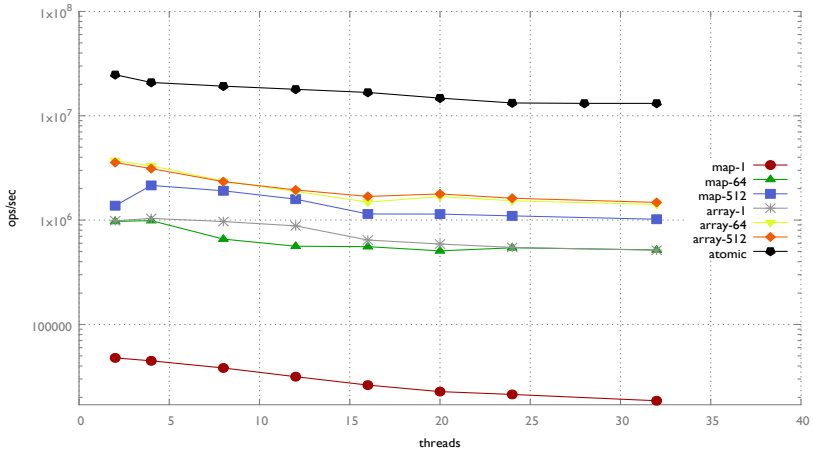


Figure 19: Atomic counter vs CRDT map vs CRDT array

Figure 19 shows that the array scales better when the merge-interval is large. However, the size of the array must be fixed to the number of threads. The implementation using the standard C++ map has an overhead on insertions when compared with the array one because it is implemented using a binary search tree form. The map has O(logN) complexity for insertions vs O(1) on the array.

In both cases the merge operation is expensive because it requires to iterate over the entire replicas, negating the benefit achieved by the asynchronous local increment.

5.3 ADD-ONLY BAG

A Grow-only Bag or Add-only Bag [4] can be implemented using a multi-headed list as shown in Figure 20.

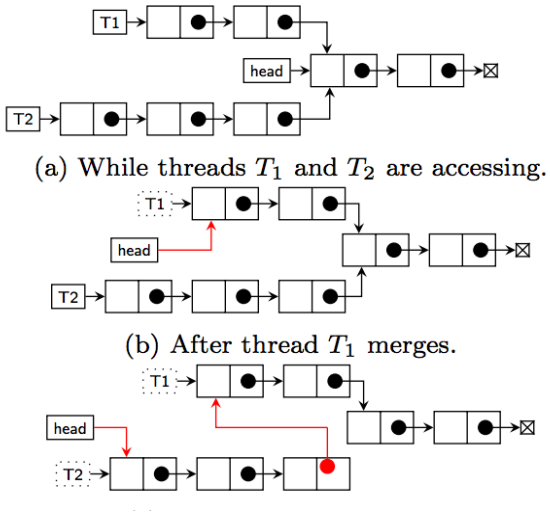


Figure 20: Mergeable grow-only bag

Each thread-local view consists of a pointer to a local head, and the global view is a pointer to a global shared head.

Insertions are performed to the head: *weakAdd* is performed to the thread local head and *strongAdd* is performed to the global head. Merge updates the global head of the list by adding the local pending elements to the global list by, synchronously, adjusting the head pointer's and not changing other threads local views. *strongLookup* traverses the list starting from the local head and if the element is not found the search will proceed on the global view.

```
1  template <typename V= int> class gBag {
2      private:
3      //global state
4      node<V> * globalHead = NULL;
5
6      //thread-local state
7      boost::thread_specific_ptr< node<V>* > localHead;
8      boost::thread_specific_ptr< node<V>** > localGraft;
9      boost::thread_specific_ptr<bool> hasValuesLocally;
10
11     //class auxiliars
12     mutex m;
13
```

```
14      public:
15      void weakAdd(const V value){
16          node<V>* tempNode = new node<V>;
17          tempNode -> payload = value;
18          tempNode -> next = *localHead;
19
20          if (!(*hasValuesLocally)){
21              *localGraft = &(tempNode -> next);
22              *hasValuesLocally = true;
23          }
24          *localHead = tempNode;
25      }
26
27      void strongAdd(const V value){
28          node<V>* tempNode = new node<V>;
29          tempNode -> payload = value;
30
31          m.lock();
32          tempNode -> next = globalHead;
33          globalHead = tempNode;
34          m.unlock();
35
36      }
37
38      bool strongLookup(const V value) {
39          if(*hasValuesLocally && searchLocally(value)) return true;
40
41          m.lock();
42          node<V>* tempNode = globalHead;
43          while(tempNode != NULL){
44              if(tempNode -> payload == value ) {
45               m.unlock();
46               return true;
47              }
48              tempNode = tempNode -> next;
49          }
50          m.unlock();
51
52          return false;
53      }
54
55      void merge() {
56          if(*hasValuesLocally) { //we may not performed a weak add
57
58            m.lock(); // protect global head accesses
59            **localGraft = globalHead;
60            globalHead = *localHead;
```

```
61            m.unlock();
62
63            *hasValuesLocally = false;
64            *localHead = NULL;
65          }
66      }
```

As mentioned in Section 4.1.2, there is a particularity with this implementation since we have no local snapshot. The semantics of the data type can lead us to scenarios where it will be expensive to replicate the entire data structure to memory due to high number of elements. On the other hand, lookups will be expensive because only *strongLookup* could be applied. We managed that by looking first in the local pending elements to avoid synchronization and only later on the globally shared elements. This way multiple threads can update the bag without copying the entire list to its thread-local storage.

We benchmarked our Add-only bag in a scenario where each thread adds sequentially 1 million elements to the bag and at every 50k additions performs a *strongLookup*, searching for the mid element(500k). Adds are performed weakly and with different frequencies of synchronization. The following figure shows how our Add-only bag behaved in the previous scenario when compared with a synchronous version:

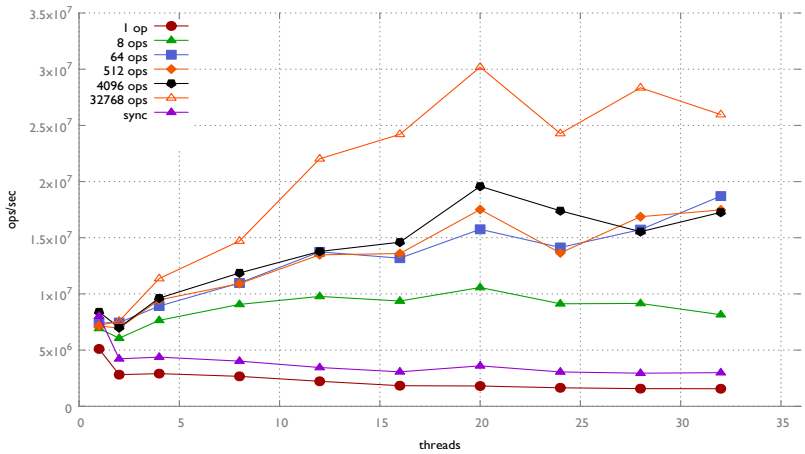


Figure 21: gBag vs sync gBag

As we can see from the results, our Add-only bag behaves well in a scenario with mostly writes, where we have regular additions to the bag and some searches for elements. We can observe some problems when scaling with shorter frequencies of synchronization. After some investigation, we concluded that the cause was our merge operation, which has only two attributions under a regular mutex, and shouldn't be causing by itself the bottleneck.

We noticed that we where paying the price of using a mutex in a scenario were we have a high-frequency operation holding the lock for short periods, leading to constant context switching which is expensive and requires time. Based on this, we decided to give it a try with spin-locks, which are a stubborn implementation of locks. When a thread fails to acquire a lock because another thread already has it, instead of going to sleep the thread will actively try to acquire it until succeeds.

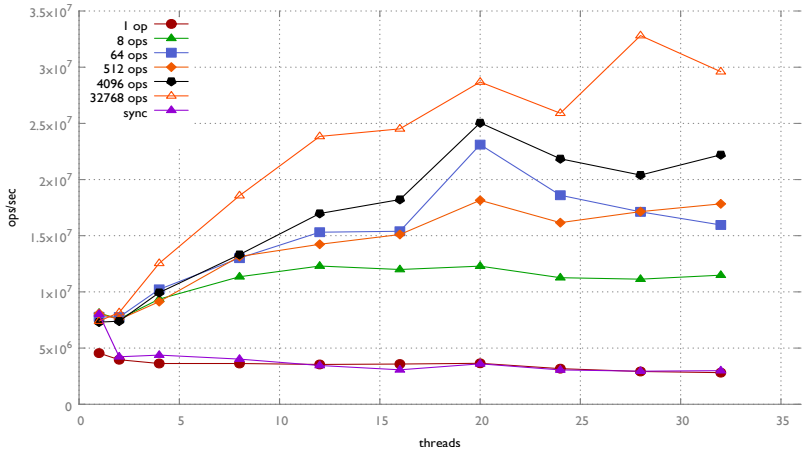


Figure 22: G-Bag vs sync G-Bag

The results show that the change to a spin-lock actually made difference in frequencies of synchronization other than 32768. For the 4096 synchronization frequency with 20 threads pushed the throughput from around 2 million of operations to around 2.5 million. Moreover, the 64 synchronization frequency achieved better results in the second experiment when compared with the 512 frequency.

5.4 OR-SET

A set can be implemented using a form of binary search tree while providing the semantics of an Observed-Removed Set with an Add-Wins policy[4]. Tree node's will be versioned using a list of *VersionInfo*, were each *VersionInfo* is composed by three elements: *first* denotes the version when the element was first added, *last* denotes the version when the element was last added, *removed* denotes the version when the element was removed.

Removed nodes will stay present on the tree but marked as removed. When an element is re-added after a removal, a new *VersionInfo* is added to the list.

Given the described structure, the state of our OR-Set will be represented by the following:

```
template <typename V = int> struct node{
  V value;
  list< std::tuple<int,int,int> > vinfo; //first,last,remove
  node * left;
  node * right;
};

template <typename T = int> class orset{
private:

  //global state
  int globalVid;
  node<T> *root;

  //thread-local state
  boost::thread_specific_ptr<int> localVid;
  boost::thread_specific_ptr< list<T> > localAdd;
  boost::thread_specific_ptr< list<T> > localRemove;
 ...
}
```

Our global state is composed by *root* that is the root of our shared global binary tree and *globalVid* that represents the global version id. In the local state we have: *localVid* that is a local snapshot of the version id, *localAdd* that is a list with the locally added elements and *localRemove* that similarly represents the locally removed elements.

The update and merge operations implementation is presented as follows:

```
void weakAdd(T value){
    (*localAdd).push_front(value);
}

void weakRemove(T value){
    (*localRemove).push_front(value);
    removeFromLocalAdded(value);
}

void strongAdd(T value){
    m.lock();
    addHelper(value);
    m.unlock();
}

void strongRemove(T value){
    m.lock();
    removeHelper(value);
    m.unlock();
```

```
20 }
21
22 void merge(){
23     m.lock();
24     int newVid = globalVid + 1;
25
26     //merge weak adds
27     for(T value: *localAdd){
28         addHelper(value, newVid);
29     }
30     (*localAdd).clear();
31
32     //merge weak removes
33     for(T value: *localRemove){
34         removeHelper(value, newVid);
35     }
36     (*localRemove).clear();
37
38     globalVid = newVid;
39     *localVid = globalVid;
40     m.unlock();
41 }
42
43 void addHelper(T value, int vid){
44     node<T>* n = searchNode(root, value);
45     if(n==NULL){
46         //element does not exist, insert with VersionInfo(vid,vid,0)
47         insert(value, root, vid);
48     }
49     else{
50         if(get<2>((n -> vinfo).front()) > 0){
51             //element exists but marked as removed, insert new VersionInfo
52             (n -> vinfo).push_front(make_tuple(vid,vid,0));
53         }
54         else{
55             //element exists and it is valid, update last
56             get<1>((n -> vinfo).front()) = vid;
57         }
58     }
59 }
60
61 void removeHelper(T value, int vid){
62     node<T>* n = searchNode(root, value);
63     if(n != NULL){
64       //check if is not marked as removed
65       if(get<2>((n->vinfo).front()) == 0){
66         //mark the element as removed with the current vid
```

```
67          get<2>((n->vinfo).front()) = vid;
68        }
69      }
70  }
```

weakAdd adds an element to the local added list and *weakRemove* similarly adds an element to the local removed list, removing the element (if present) of the local added list.

merge will synchronously increment the global version id and iterate both local removed and added lists, calling the respective helpers for each element. The *addHelper* will handle the logic to add an element to the tree, it starts by checking if the element already exists and if not, inserts the element in the tree at the correct position adding the respective *VersionInfo* to its list $(newVid, newVid, \emptyset)$. If the element exists, validates if the element is a removed one by checking if the head *VersionInfo removed* is not $\emptyset$. If it is a removed one inserts a new *VersionInfo*, if not updates the head *VersionInfo last*. The *removeHelper* will similarly handle the logic to remove an element from the tree, by simply marking the element as removed if the element exists and is not already marked as removed.

```
1
2  bool weakLookup(T value){
3      //checks if was locally added
4      for(T localValue: *localAdd){
5        if(localValue == value) return true;
6      }
7
8      //searches on the shared copy
9      m.lock();
10     node<T>* n = searchNode(root, value);
11     if(n == NULL) {
12         m.unlock();
13         return false;
14     }
15     list<std::tuple<int,int,int>> vinfo (n -> vinfo);
16     m.unlock();
17
18     for(auto v : vinfo){
19       //not marked as removed and first added in an observable version
20       if(get<2>(v) == 0 && get<0>(v) <= *localVid) return true;
21
22       //first added at an observable version and removed at a concurrent/
             previous version
23       else if(get<0>(v) <= *localVid && *localVid < get<2>(v)) return true;
24     }
25
26     return false;
```

```
27 }
28
29 bool strongLookup(T value){
30     //checks if was locally added
31     for(T localValue: *localAdd){
32       if(localValue == value) return true;
33     }
34
35     //searches on the shared copy
36     m.lock();
37     node<T>* n = searchNode(root, value);
38     if(n == NULL) {
39         m.unlock();
40         return false;
41     }
42     list<std::tuple<int,int,int>> vinfo (n -> vinfo);
43     m.unlock();
44
45     //validates if the element is not marked as removed
46     for(auto v : vinfo){
47       if(get<2>(v) == 0) return true;
48     }
49
50     return false;
51 }
```

As in the *gBag* implementation we will not have an fully local snapshot of the set. On the other hand, we will leverage the semantics of the data type to perform the *weakLookup* with the global copy of the set and the local snapshot of the vid. The lookup starts by checking if the element was locally added if so returns true. Then, it searches for the element synchronously in the globally shared copy and if it exists checks if the head *VersionInfo* is valid according to *localVid*. An element is valid if *removed* is $\emptyset$ and the element was first added in a version lower or equal to what we can observe, or if it was first added in a version lower or equal to what we can observe and if was removed in a previous/concurrent version(*Add-wins*).

strongLookup is similar but will only check if the head *VersionInfo* of the element is not $\emptyset$.

We have benchmarked our OR-Set in a scenario where each thread alternates between adds and removes, in a total of 100k operations. A remove is performed for every 10 adds. At each 5k operations, we perform a *strongLookup*, searching for the mid element(50k). Adds and removes are performed weakly and merge is performed within different frequencies of synchronization. We have also used spin-locks because, as in gBag, the shorter frequencies of synchronization were not scaling with the number of threads. The following

figure shows how our OR-Set behaved in the previous scenario when compared with a synchronous version:

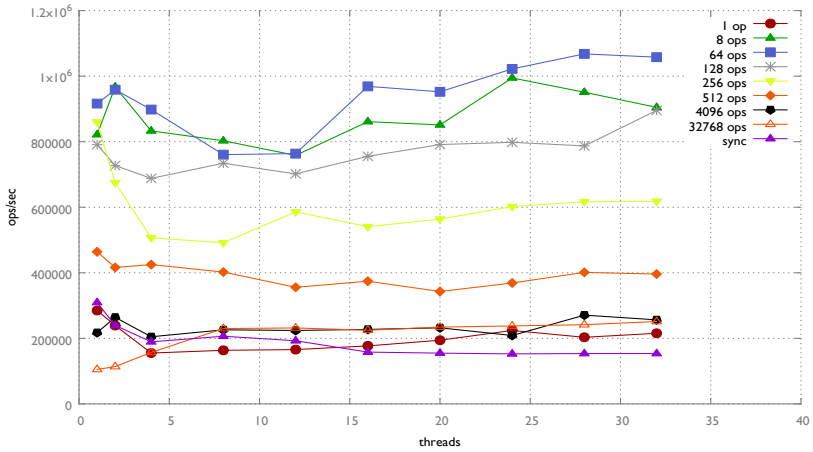


Figure 23: OR-Set vs sync OR-Set

Mostly all the synchronization frequencies outperformed the synchronous version. We can also observe that the 64 frequency of synchronization outperformed all the others, followed by the 8 frequency. Moreover, from the 128 to the 32768 frequency as short the frequency is better is the throughput. This is an indicator that merging more elements (slower merges), were impacting the performance more than merging fewer elements more frequently.

We have also implemented an alternative version of our OR-Set, where we didn't delegate all the work to merge time and instead we did it while performing weak operations:

```
1  boost::thread_specific_ptr< list< pair<T, node<T>*> > > localAdd;
2  boost::thread_specific_ptr< list< pair<node<T>*, int> > > localRemove;
3
4
5  void weakAdd(T value){
6      m.lock();
7      //finding an estimated position for the insertion(possible parent)
8      node<T>* parent = findParentNode(root, value, root);
9      m.unlock();
10
11     //storing the estimated position with the value
12     (*localAdd).push_front(make_pair(value, parent));
13 }
14
15
```

```
16 void weakAddHelper(T value, node<T>* possibleParent, int vid){
17     //start searching from the possible parent
18     node<T>* n = searchNode(possibleParent, value);
19     if(n == NULL){
20         //insert starting on the possible parent
21         insert(value, possibleParent, vid);
22     }
23     else{
24         if(get<2>((n -> vinfo).front()) > 0){
25             (n -> vinfo).push_front(make_tuple(vid,vid,0));
26         }
27         else{
28             get<1>((n -> vinfo).front()) = vid;
29         }
30     }
31 }
32
33 void weakRemove(T value){
34     m.lock();
35     node<T>* n = searchNode(root, value);
36
37     //only remove if we can observe it
38     if(n != NULL && (get<2>((n -> vinfo).front()) == 0)){
39       int last = get<1>((n -> vinfo).front());
40       (*localRemove).push_front(make_pair(n,last));
41     }
42     m.unlock();
43
44     removeFromLocalAdded(value);
45 }
46
47 void weakRemoveHelper(node<T>* node, int observedLast, int vid){
48   //check if was not removed by other threads meanwhile
49   bool notConcurrentlyRemoved = get<2>((node -> vinfo).front()) == 0;
50
51   //check if was not updated by other threads meanwhile(add-wins)
52   bool notConcurrentlyUpdated = get<1>((node -> vinfo).front()) ==
       observedLast;
53
54   if(notConcurrentlyRemoved && notConcurrentlyUpdated){
55       get<2>((node -> vinfo).front()) = vid;
56   }
57 }
```

When *weakAdd* is performed, the potential parent node is identified and this information is stored on *localAdd*. In merge time, when the *weakAddHelper* is called, we search for the

element that we want to insert starting from the possible parent since our set is a binary tree. Then if the element does not exist, we will insert it starting from the possible parent. The remaining logic is the same than in the previous implementation.

When *weakRemove* is performed, we search for the element and if we can observe it, we will add it to *localRemove* with the head *VersionInfo last*. Then in merge time, the *weakRemoveHelper* will check if other threads did not remove the element by checking the *VersionInfo* head *removed* and if was not updated, by checking if the head *VersionInfo last* is not different from what was observed when the weak operation was performed. If verified these conditions, the element is marked as removed.

We benchmarked this version of our OR-Set in the same scenario. The following figure presents the results:

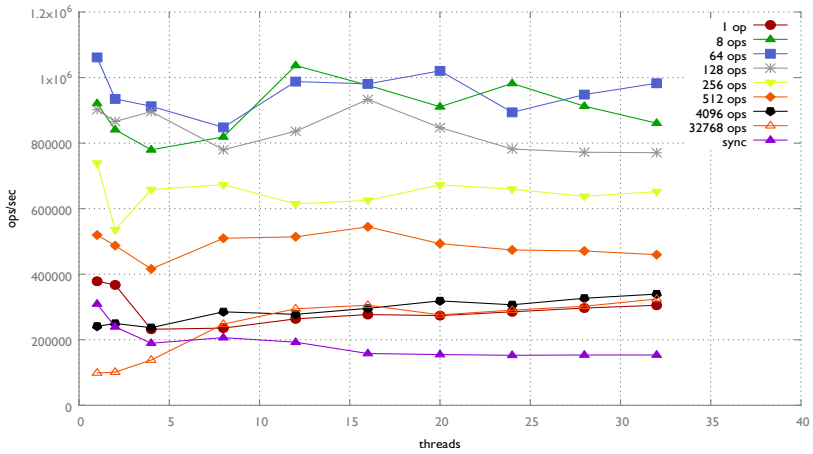


Figure 24: OR-Set v2 vs sync OR-Set

As we can see from the results we were able to improve the throughput of the 8 synchronization frequency being almost side by side with the 64 frequency. The other frequencies also slightly improved in terms of performance.

5.5 QUEUE

A hybrid mergeable queue, as proposed in Section 4.1.4, can be implemented using a singly-linked list similar to a linearizable queue. The items enqueued are added to the tail of the list, while dequeue is performed from the head.

```
1 //global-state
2 node<V>* globalHead = NULL;
3 node<V>* globalTail = NULL;
```

```
4
5 //thread-local state
6 boost::thread_specific_ptr< node<V>* > localHead;
7 boost::thread_specific_ptr< node<V>* > localTail;
8
9 //class auxiliars
10 mutex m;
11
12 void weakEnqueue(V value){
13     if(*localHead == NULL){
14         *localHead = new node<V>;
15         (*localHead) -> payload = value;
16         (*localHead) -> next = NULL;
17         *localTail = *localHead;
18     }
19     else{
20         node<V> * tempNode = new node<V>;
21         tempNode -> payload = value;
22         tempNode -> next = NULL;
23         (*localTail) -> next = tempNode;
24         *localTail = tempNode;
25     }
26 }
27
28 V strongDequeue(){
29     m.lock();
30     if(globalHead == NULL){
31         m.unlock();
32         return V(EMPTY_QUEUE); //value representing an empty queue
33     }
34     else{
35         V res = globalHead -> payload;
36         node<V>* n = globalHead -> next;
37         globalHead = n;
38         m.unlock();
39         return res;
40    }
41 }
42
43 void merge(){
44     if(*localTail == NULL) return;
45
46     m.lock();
47     if(globalHead != NULL) {
48         globalTail -> next = *localHead;
49         globalTail = *localTail;
50         m.unlock();
```

```
51      }
52      else {
53          globalHead = *localHead;
54          globalTail = *localTail;
55          m.unlock();
56      }
57
58      *localHead = NULL;
59      *localTail = NULL;
60  }
```

A mergeable queue instance contains a global view – (*globalHead, globalTail*), which points to the head and tail nodes respectively of a *global list* and a local view – (*localHead, localTail*), which are the head and tail of a thread's *local list*. The local list collects the items enqueued by a thread (*weakEnqueue*) that are not yet merged. The *merge* atomically appends the local list to the global list. The *dequeue* operation directly updates the shared part of the list. For some data types, an update on the shared part of the data structure should preserve the old version, because local views may be keeping a reference to it. However, there is no *weakRead* defined that needs to observe a version before a concurrent dequeue. Hence, there is no need to keep those versions, which simplifies the implementation.

We have benchmarked our queue in a scenario where each thread alternates between enqueues and dequeues, in a total of 1 million operations. A dequeue is performed for every 1k enqueues. The following figure shows how our hybrid mergeable queue behaved in the previous scenario when compared with a *multiple-reader/multiple-writer* lock-free queue from boost:

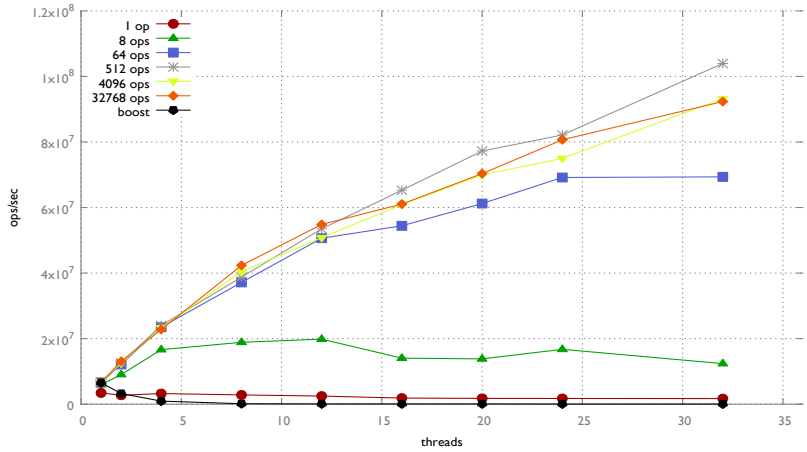


Figure 25: Hybrid mergeable queue vs boost lock-free queue

The time needed to merge a group of nodes is the same as the time needed to enqueue a single node. By batching the enqueues, we can reduce the number of synchronization operations, thus improving the overall throughput. In the particular scenario of the benchmark, we achieved better throughput when performing enqueues in batches of 512, being this our sweet spot.

Surprisingly we achieved better performance when synchronizing in every operation than with boost's lock-free queue.

5.6 ELECTRONIC VOTE

In this section, we will present a *time-bounded* hybrid MDT, based on a direct use-case application. Consider a scenario where we have an electronic voting system, for an election, where we have multiple candidates. The system consists of a machine receiving votes from multiple connections and each connection is handled by a thread.

At the beginning of the election, threads can receive multiple votes, process them without synchronizing with other threads and later merge the results when needed. Allowing to achieve partial results without impacting the global performance of the system.

When the voting time is ending, a more precise approach could be used to get concrete results by switching to synchronous operations. To satisfy these requirements, we have built a hybrid mergeable E-Vote:

```
1  //global state
2  std::atomic<bool> globalCanVote;
3  std::atomic<bool> globalSwitchOn;
4  std::atomic<int> * globalCandidates;
5
6  //thread-local state
7  boost::thread_specific_ptr<bool> localCanVote;
8  boost::thread_specific_ptr<bool> localSwitchOn;
9  boost::thread_specific_ptr<std::vector<int>> localCandidates;
10 boost::thread_specific_ptr<std::vector<int>> localSnapshotCandidates;
11
12 //class auxiliars
13 std::atomic<int> mergeTimeMillis;
14 int candidateNumber = 0;
15
16 void voteClose(){
17     boost::this_thread::sleep(boost::posix_time::milliseconds(DURATION));
18     globalCanVote=false;
19 }
20
21 void activateSwitch(){
```

```
22      boost::this_thread::sleep(boost::posix_time::milliseconds(THRESHOLD));
23      globalSwitchOn=true;
24  }
```

An array of integers is used to represent our election candidates, where each entry accumulates the number of votes for a candidate. There is a global shared version of the array (*globalCandidates*), and two thread-local ones (*localCandidates* and *localSnapshotCandidates*). There are also two different flags: *canVote* which indicates whether the voting is still open or not; and *switchOn* which indicates if we should switch to synchronous operations.

As this is a time-based MDT, we have two threads responsible to change *globalCanVote* and *globalSwitchOn* when in the respective time. The thread-local versions of these flags are updated based on the changes to the global ones at merge time.

```
1  void weakVote(int id){
2     if(id < (candidateNumber) ) (*localCandidates)[id]++;
3  }
4
5  void strongVote(int id){
6     if(id < (candidateNumber)) globalCandidates[id]++;
7  }
8
9  void vote(int id){
10    if(!(*localSwitchOn)){
11       weakVote(id);
12    }
13    else strongVote(id);
14 }
15
16 void merge(){
17    //merging
18    for(int i=0; i<candidateNumber; i++){
19       globalCandidates[i] += (*localCandidates)[i];
20    }
21
22    //updating the snapshot
23    for(int i=0; i<candidateNumber; i++){
24      (*localSnapshotCandidates)[i] = globalCandidates[i];
25    }
26
27    //reseting thread-local state
28    localCandidates.reset(new std::vector<int>(candidateNumber));
29
30    if(globalSwitchOn) *localSwitchOn=true;
31    if(!globalCanVote) *localCanVote=false;
```

```
32
33 }
```

weakVote increments the position of a given candidate on the *localCandidates* and *strongVote* does the same on *globalCandidates*; *vote* is wrapping the use of *strongVote* or *weakVote* based on the *localSwitchOn* flag; *merge* starts by adding the local accumulated votes to the global copy, then updates the *localSnapshot* candidates by making a copy of *globalCandidates*. Also, resets the *localCandidates* putting all candidates votes to $\emptyset$ and checks if the thread should update the *localSwitchOn* and *localCanVote* flags.

```
1
2 std::vector<int> weakResults(){
3    std::vector<int> results(candidateNumber);
4
5    for(int i=0; i<candidateNumber; i++){
6       results[i]= (*localSnapshotCandidates)[i] + (*localCandidates)[i];
7    }
8
9    return results;
10 }
11
12 std::vector<int> strongResults(){
13    std::vector<int> results(candidateNumber);
14
15    for(int i=0; i<candidateNumber; i++){
16       results[i]= globalCandidates[i] + (*localCandidates)[i];
17    }
18
19    return results;
20 }
```

weakResults can be achieved by adding the locally added votes(*localCandidate*) to the most recent copy from global state (*localSnapshotCandidates*), not impacting the work other threads are doing. *strongResults* can be achieved by directly Reading from *globalCandidates* and adding the pending local operations.

Our E-Vote MDT was benchmarked in a scenario with five candidates and votes randomly distributed between them. The election lasts for 60 seconds with a defined threshold of 30 seconds to switch to synchronous operations. While performing weak operations threads synchronized using time intervals. The following plot shows how our implementation behaved with different synchronization intervals when compared with a synchronous version (only strong operations):

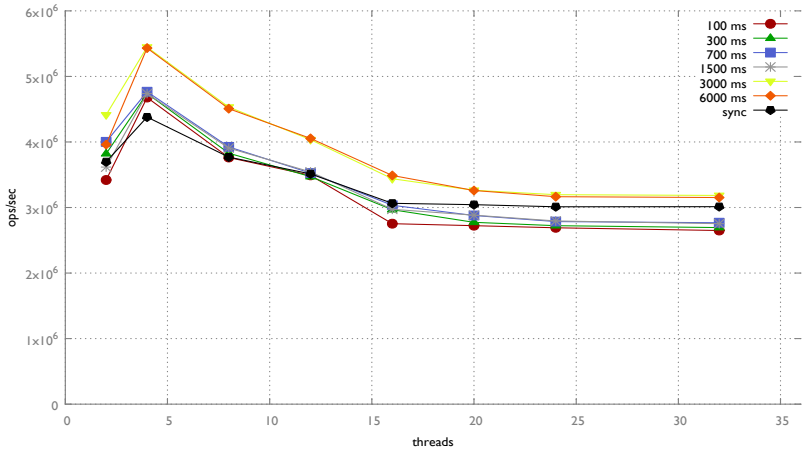


Figure 26: Hybrid e-vote version 1 vs sync

The results were not the expected, the number of votes per second was not growing with the number of threads and synchronization frequency. At some points, the synchronous version outperformed all the other versions.

We recognized that in this particular time-bounded MDT we had a problem that we didn't realize until now. Threads were syncing based on time, which means that all threads were merging almost at the same moment causing a slowdown. The solution passed by making threads within the same frequency of synchronization to sync in different time instants. The results of this change are notable:

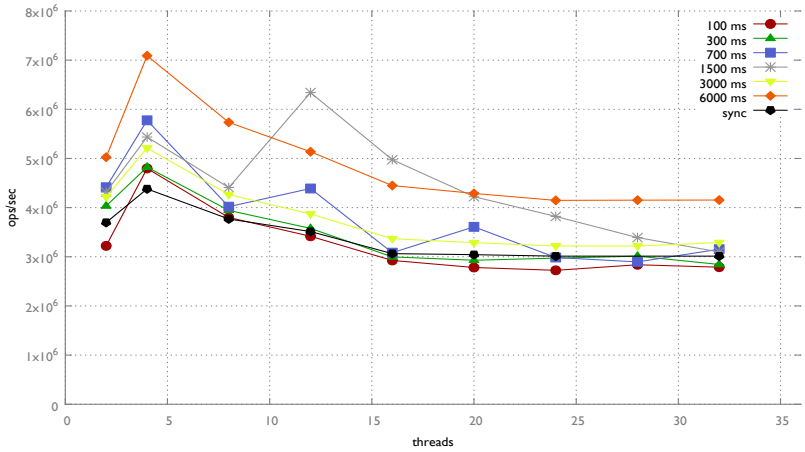


Figure 27: Hybrid e-vote version 2 vs sync

Although the throughput doesn't grow with the number of threads, the current results look much better when compared with the previous version. We now have several frequencies outperforming the synchronous(atomic) one.

Notice that, we are not making any reads on the experience, but we can't take advantage of our *weakRead* and obtain preliminary results without significantly impacting performance. Also switching to strong operations later in time would improve the throughput.

Until now, we have been using strong operations either when we need to achieve a given number of operations or when the semantics of the data types require operations to be synchronous. In the presented case, strong operations are directly tied to the use-case, we switch to them based on a time threshold. Time-based approaches open the door for different MDTs applications from the ones presented so far.

5.7 SUMMARY

This chapter started with the formulation of some theoretical questions, and we have answered them. We have compared a thread-local approach with some of the existent concurrency mechanisms on the C++ language and the results in the tested scenarios showed that our version outperformed the other mechanisms. Also, performance was very close to a scenario where we don't use any synchronization mechanism at all.

We have also built two P-Counters as CRDTs for shared memory and verified that the approach is, in fact, inefficient for the environment when compared to MDTs, due to unnecessary logic on the merge operation.

From the several benchmarks to our MDTs we have also observed that the mergeable versions are indeed more efficient than the respective synchronous versions. MDTs can be applied in situations where divergence between threads is not allowed by using only strong operations, resulting in the linearizable versions. Even more, we can leverage from identifying situations where more relaxed approaches can be used and then switch to versions where no divergence is allowed. We did it using different types of thresholds: number of operations and time. With the eVote, we demonstrated that MDTs can be applied on a real-world use-case.

6

FINAL CONSIDERATIONS

6.1 FINAL CONSIDERATIONS

In this dissertation, we migrated an existent concept of data types on distributed systems to shared memory. The idea consists of relaxing synchronization by letting the different participants of a system perform operations asynchronously and later when needed, synchronize with other participants (without conflicts) converging to the expected result.

Our contribution started by developing the existent concept of *Mergeable Data Type* by studying the literature and by exploring the needs of a shared memory environment. Leading to a programming model that we named *Global-Local View* and later to the definition of a formal specification for MDTs. An immediate distinction between two types of MDTs emerged: purely and hybrid. A purely MDT only has defined weak operations while a hybrid MDT has both weak and strong operations. Combining weak and strong operations we have the flexibility to build data types with customizable semantics on the spectrum between sequential and purely MDTs on a trade-off between update visibility and performance.

Using the specification we have defined a portfolio with a range of data types from counters, queues, and sets to an electronic election use case. Finally, we have added a practical component to this work where we have implemented each of the data types in our portfolio using a design pattern created by us that resulted from our programming model. The results demonstrate that in the appropriated scenarios we can largely benefit from avoiding synchronization between threads and still achieve precise results without degrading performance with simple and user-friendly implementations.

All the code used is open-source and published under MIT license.

6.2 FUTURE WORK

As future work, the GL-View model could be extended to other applications that would enrich our current portfolio of MDTs.

A work-stealing queue is used to distribute tasks among threads running in parallel. In Cilk runtime [14] each thread owns a queue with operations *pushTop*, *popTop*, and *popBottom*. There is no *pushBottom*. When a thread is devoid of tasks, it retrieves one from its queue using *popTop*, executes it and may generate new tasks that are added to its queue using *pushTop*. When a thread's task queue is empty, it steals from other thread's queue using *popBottom*. A work stealing queue with this semantics is a natural fit for the global-local view model. Instead of a queue per thread, we have a multi-view queue with a global view and a local-view per thread. *pushTop* and *popTop* executes on the thread-local views, and *popBottom* on the global view. One downside of this design is that it may prevent threads from stealing tasks when the global view is empty even if there are unmerged tasks in the local views. To avoid this, threads can be forced to merge when the global view drops below a threshold.

Another possible application are in-memory multi-core databases. In high contention workloads, we can achieve high performance by allowing concurrent transactions to proceed in parallel on different cores. Instead of serializing the access to objects, transactions can update a per core copy of the object and merge them later. In [32], authors describe a system that automatically parallelizes contending transactions if all modifications to contending items are commutative. A multi-view data type implemented in the global-local view model is a natural fit to such a scenario.

Moreover, some of the presented data types on this document would fit directly on existent applications. Message queues where multiple messages can be batched together and added to a shared queue is a direct application of our hybrid queue. Applications that use aggregation counters that are computed by parallel threads can use our mergeable counter. Similarly, objects that store statistical measures such as sums, min, max etc. that are computed by parallel threads can also benefit from the global-local view model.

BIBLIOGRAPHY

[1] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards consistency oblivious programming. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 65–79, 2011.

[2] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010.

[3] Deepthi Devaki Akkoorath and Annette Bieniusa. Transactions on mergeable objects. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 427–444, 2015.

[4] Deepthi Devaki Akkoorath and Annette Bieniusa. Highly-scalable concurrent objects. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 13:1–13:4, 2016.

[5] Deepthi Devaki Akkoorath, José Brandão, Annette Bieniusa, and Carlos Baquero. Global-local view: Scalable consistency for concurrent data types. *CoRR*, abs/1705.03704, 2017.

[6] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 62–76, 2015.

[7] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks and multiprocessor coordination. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 348–358, 1991.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, page 7, 2000.

[9] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 691–707, 2010.

[10] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 568–590, 2015.

[11] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 187–200, 2014.

[12] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, pages 194–208, 2006.

[13] Brijesh Dongol and Lindsay Groves. Towards linking correctness conditions for concurrent objects and contextual trace refinement. In *Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015.*, pages 107–111, 2015.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223, 1998.

[15] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[16] Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 388–402, 2003.

[17] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 207–216, 2008.

[18] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 92–101, 2003.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300, 1993.

[20] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[21] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[22] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 31:1–31:16, 2016.

[23] C Kirsch, Hannes Payer, and Harald Röck. Scal: Non-linearizable computing breaks the scalability barrier. Technical report, Technical Report 2010-07, Department of Computer Sciences, University of Salzburg, 2010.

[24] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 30–39, 2014.

[25] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[26] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 168–183, 2015.

[27] Paul E McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. Rcu usage in the linux kernel: one decade later. *Technical report*, 2013.

[28] Paul E. Mckenney and John D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[29] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Grasso Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 69–80, 2007.

[30] Mark Moir. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings*, pages 305–319, 1997.

[31] Mark Moir and Nir Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications.* 2004.

[32] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Tappan Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 511–524, 2014.

[33] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400, 2011.

[34] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213, 1995.

[36] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 682–696, 2016.