An Experimental Evaluation of Value-Splitting in Transactional Memory Systems*

Rui Ribeiro[0009−0002−3421−8932], Nuno Faria[0000−0003−4691−0440], and José Pereira[0000−0002−3341−9217]

INESC TEC and U. Minho, Braga, Portugal

Abstract. Numeric values can turn into hot-spots when being modified by many concurrent operations, degrading performance. Previous research on the topic addresses the problem with the usage of splitting techniques, such as Multi-Record Values (MRVs) and Phase Reconciliation (PR). As such, their application in transactional memory could prove useful. However, these techniques assume either the usage of a database engine or having a fixed number of concurrent tasks, usually processor cores, running at any given moment, which do not directly fit transactional memory systems. In this work, we propose adaptations of both MRVs and PR to a software transactional memory system, discussing how each assumption can be met and each mechanism implemented. Experiments show under what conditions each of these techniques is feasible and results in improving performance.

Keywords: Concurrency hot-spots · Transactional memory · Value-splitting

1 Introduction

Programs written in a typical sequential fashion are not able to take advantage of the power of multiple cores. Instead, parallel/concurrent programming techniques must be employed, such as the usage of threads. These, in turn, require additional coordination, so that multiple threads do not attempt to concurrently modify the same memory locations and create inconsistent data. Mutual exclusion (or locking) is one such synchronisation mechanism, in which a thread must acquire a lock before accessing its respective data. This approach has some well-known limitations, such as being susceptible to deadlocking, not being composable, and requiring additional effort to be implemented correctly.

Transactional memory (TM) [8] emerges as a programming paradigm well suited to multi-core systems, avoiding the complexity and error-proneness of locking mechanisms. These problems do not disappear per se; they are instead passed on to the developers of the TM system. Due to this, TM implementations have the responsibility of being both correct and performant. As is the case with

* This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

mutual exclusion, extra care must be taken in order to avoid hot-spots. These can arise in a variety of situations, such as online shopping, social media counters, and inventory management applications.

In this work, we analyse the applicability of value-splitting techniques (Multi-Record Values [5] and Phase Reconciliation [15], in particular) to transactional memory systems and evaluate our own adaptations with several synthetic tests.

2 Background

In general terms, value-splitting is the process of dividing a single value into multiple chunks so that multiple processes/workers are able to work in parallel without creating contention or causing conflicts. The two techniques we analyse achieve this in different ways, in regards to their storage/assignment of chunks and their internal adjustment to varying workloads. In this work, we are only considering non-negative integer values as our splitting target. None of these techniques weakens consistency, as parallel operations are allowed only as long as they commute and read operations always return the total current value.

2.1 Multi-Record Values

Multi-Record Value (MRV) [5] is a novel technique that aims to mitigate performance penalties and conflicts that arise in hot-spots in both centralised and distributed database systems. MRVs do this by splitting contended values across multiple records and then using random numbers to pick the record which will be accessed, ensuring that the updates are uniformly spread.

The number of records designated to hold the value is dynamically adjusted according to the workload. Thus, it is possible for the MRV to never merge back into a single value if the contention on it is high enough. This does not pose a consistency problem since MRV operations can be performed independently of the number of records. To read the real value of an MRV, the transaction performs a sum of all partial amounts.

There are two main insights presented regarding the assignment and management of the records:

- *Clients are not statically assigned to records.* Instead, a random number is generated each time a client wants to access the MRV. This ensures an even spread of accesses and avoids explicit coordination between clients;
- *The number used for looking up a record is different from the number used as the key of the record.* Instead, the records are stored in a ring-like structure and a constant N is selected as the upper bound for the number of records that can exist. To perform a lookup, the algorithm generates a random number and chooses the closest record that follows it, looping back to the beginning in case it does not find any. This reduces overhead, as the number of existing records does not need to be stored or counted.

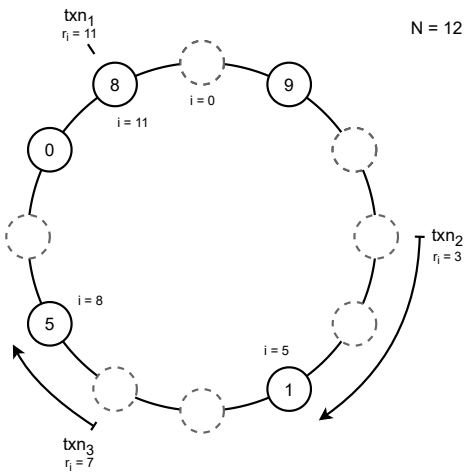


Fig. 1: Example of an MRV lookup.

As an example, we will show how an addition is performed using the MRV structure shown in Fig. 1. First, we pick a random integer in the $[0, N-1]$ interval. Then, we lookup the position determined by our random index and check if it has an initialised record. If it does not, we keep checking the following positions until we find a record. Upon finding a record, we can perform the addition as normal. For instance, txn_1 found a record on the exact position that it had randomly picked ($r_i = i = 11$), while both txn_2 ($r_i = 3$) and txn_3 ($r_i = 7$) needed to traverse to $i = 5$ and $i = 8$, respectively.

Subtractions start out in a similar fashion to additions. However, upon finding an initialised record, we compare the stored value to the one we subtract before performing the operation. If it is greater than or equal, we can perform the subtraction as usual. If it is not, we keep looking up other records until either the aggregated value is enough, resulting in a successful operation, or we loop back to our starting record, resulting in an unsuccessful operation.

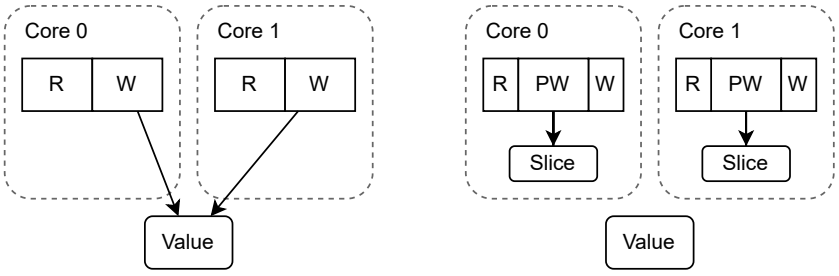
2.2 Phase reconciliation

Phase reconciliation (PR) [15] is a concurrency control technique for in-memory transactions, targeting workloads where small subsets of items are subject to a large number of updates. In addition to numeric values, phase reconciliation is also designed to work on more complex data structures, such as ordered tuples and top-K sets, which we do not consider in this work.

Doppel, the phase reconciliation database introduced in the same paper, cycles through three distinct execution phases: the *joined* phase, the *split* phase, and the *reconciliation* phase. Phase cycles are specific to single data items, i.e., one item can be in a joined phase while another distinct item is in a split phase.

The *joined* phase uses a typical optimistic concurrency control (OCC) protocol and allows any kind of transaction to be executed. Once data contention reaches a level of unnecessary serial execution, the system switches to the *split* phase. Records marked as "split" are divided between cores and only a reserved operation, the one where the contention was detected, is allowed to execute on these partial values; other transactions on split items are blocked until the database returns to the joined phase. Finally, the *reconciliation* phase merges the values from the cores back into the global store and the cycle restarts.

In order to maintain correctness during the split phase, only a small subset of operations are available. When reconciling values, these operations must have the same result as if they were executed in sequential order.



(a) Joined phase. (b) Split phase.

Fig. 2: Example of a PR write.

As an example, we will show how an addition is performed using the PR object in Fig. 2, in both the joined and split phases. In the joined phase (Fig. 2a), concurrent transactions on different cores access the same value in memory and can conflict when performing an addition (overlapping write sets). If a high enough level of contention is reached, then the value switches to the split phase (Fig. 2b). While in the split state, different cores have their own slice of the value and can issue "private" writes without conflicting (the write sets no longer overlap). When the value undergoes the reconciliation phase, both private slices are added back together.

Subtractions work in a similar manner, but with the possibility of failing if the slice/value (in the split and joined phases, respectively) does not contain a sufficient amount to subtract. This is in contrast to the MRV approach, where it does not fail if a given slice is not enough but the value as a whole is, since the subtraction can be performed over multiple slices.

3 Implementation

We have implemented our adaptations as a C++ library, using Wyatt-STM [6] as our target TM system. Wyatt-STM became our target due to its rich feature

set and real-world applicability. It includes functionality such as explicit retires and aborts, custom function invocation upon commit/abort, and object-level transactional accesses.

Our common interface for MRVs and PR specifies the following operations:

- `read`: Fetches the entire value;
- `add`: Takes a value and adds it;
- `sub`: Takes a value and tries to subtract it. It can fail if the stored value (minuend) is smaller than the taken one (subtrahend).

We use various functional immutable data structures from the *immer* library [18]. Updates can be done efficiently and safely by creating a new instance that shares the physical representation of previous content. Therefore, they lend themselves quite well to concurrent scenarios and, in the particular case of transactional memory, to rolling back transactions on complex data structures.

Since our main goal is to study the feasibility of value-splitting techniques in TM systems, we have not evaluated every implementation strategy presented in the MRVs and PR papers. Instead, we chose "good enough" defaults, and, where applicable, it will be stated in this article what those defaults are, along with the explicit deviations that were made from the original research.

Our MRVs and PR adaptations are independent of each other, but both follow a similar architecture (Fig. 3). At the top level, there is an object manager, which is a singleton that stores pointers to all the active objects of its respective type. These are used by the manager's workers, the ones responsible for periodically adjusting the values to the running workload. The pointers are all contained in an immutable *immer* map, so that application threads are able to traverse through all of the objects without causing conflicts with concurrent additions/removals that result in a new instance.

To track contention on our objects, each operation that is executed logs its status upon abort or commit. For our purposes, an abort is counted whenever a transaction fails to commit, e.g., due to read/write conflicts on the same MRV partition or running out of stock on a PR partition. This includes retries, which would make a transaction that retried N times before committing count as N aborts.

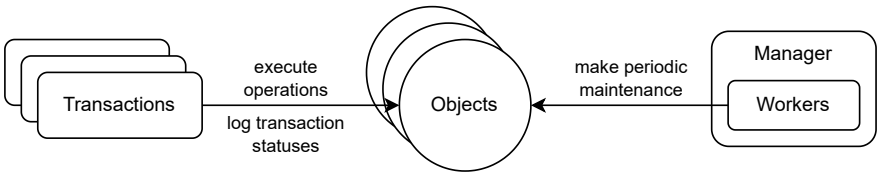


Fig. 3: Generic architecture for both techniques.

3.1 Multi-Record Values

The underlying data structure of our MRVs adaptation is quite simple: an immutable *immer* vector of transactional integers (the chunks). Making the vector itself immutable is the key to letting it be shared between threads. Since our transactions can at times traverse the entire vector (in `read` operations, for example), we needed a way to let a thread be able to iterate through all the chunks while another thread added/removed them from the vector. The transactional integers escape this immutability, since they are not reassigned, but only internally mutated. To add/remove chunks, a copy of the vector is made with the new changes applied. The *immer* library allows for the use of this functional approach without taking a substantial performance hit.

MRVs require two background workers, named *adjust* and *balance*. The adjust worker manages the dynamic growth and shrinkage of the MRVs, depending on the workload: Elements are added or removed by creating a new instance of the vector that is used to update the singleton. Care is taken to update elements being removed to force conflicts with concurrent application threads. The balance worker evens the values stored in the chunks in order to reduce contention, namely, for subtract operations that might need to access multiple chunks to complete. This naturally causes conflicts with concurrent threads accessing the same elements, thus ensuring consistency. Our workers follow the same periods as the original adjust and balance workers, which are 1000ms and 100ms, respectively.

The original paper evaluated the impact of different window sizes, that is, the interval of time that workers considered for measuring contention. We opted to discard this concept of windows and instead consider all the information that followed the last fetch. That is, we have simple counters for the total number of aborts and commits, and they are reset every time the workers fetch the data. In our case, only the adjust worker needs this data, so it is reset every 1000 ms.

We have also opted to ignore the second key insight of MRVs, which was to decouple the lookup numbers from the record keys. Due to the internal structure of our adaptation, the total number of chunks is known (the vector needs it) and can be accessed at no extra performance cost. Every element of the vector is also filled with an available transactional variable, making each lookup exact and not approximate like the original implementation.

Random number generation plays a crucial part in MRV performance, being used in additions and subtractions to lookup random records. We used a Mersenne Twister (MT) pseudo-random number generator (`std::mt19937`) per thread that feeds a uniform integer distribution. Each of the MT generators is seeded with `std::random_device` to avoid generating the same sequence of numbers on every thread.

3.2 Phase reconciliation

Compared to MRVs, PR requires a more complex structure to work. When the object is in the joined phase, it consists of a single transactional integer. When

in the split phase, it comprises a fixed operation and a vector of transactional integers, each exclusive to its assigned thread. We use a transactional boolean to indicate the object state, if it is in a split or joined phase. Since all transactions that operate on the object need to fetch this boolean, a phase change will conflict with running transactions and force a restart.

Phase transitions occur as detailed in §2.2. Note that the reconciliation phase is not treated as an explicit state in our implementation, since the split phase transitions directly into the joined phase within a single transaction, avoiding consistency issues. Reconciliation is triggered in a similar way as the adaptation used in the MRV paper [5]: it can happen if there is any client waiting (e.g., tried to perform a read and is now blocked until the joined phase) or if there was any abort due to no stock (zeroed counter).

PR has only one background worker, responsible for triggering phase transitions on the objects. Every 20ms, as in the original paper, the worker fetches the current metadata and decides whether to transition or not.

One downside of PR is static thread allocation. In particular, in our implementation, threads must register themselves with the object before any operation is executed and cannot be later unregistered. By comparison, MRVs are not bound to the number of threads that are interacting with them.

4 Experiments

Our evaluation has three targets: one whole value that will serve as a baseline (*Single*) and two other values that implement the two value-splitting techniques we presented in this article (*MRVs* and *PR*). The results were obtained through an average of five executions of 60 s each, measuring the throughput of successful commits. All tests were executed on a machine with the following specifications:

- **CPU:** 2x HiSilicon Kunpeng 920 (ARM)
- **RAM:** 8x 32GB DDR4 2666MHz
- **OS:** Rocky Linux 8.7
- **Kernel:** Linux 4.18.0
- **Compiler:** gcc 8.5.0

We created our first experiment with the goal of determining the optimal number of clients (threads that run transactions) for our tests. To do so, we created a micro-benchmark comprised of long-running transactions with singular additions to the object. To achieve the long duration, we have added artificial padding, aiming to simulate a real-world scenario where a transaction would include work other than updating our object. The results are presented in Fig. 4.

PR has the best overall performance of the three due to static thread assignment. Since the test only performs additions and no reads, the object can stay in the split phase, and each client can effectively work independently of the others. MRVs do not fall too far behind, also delivering better performance compared to the baseline case. The single-value approach does not scale as transactions

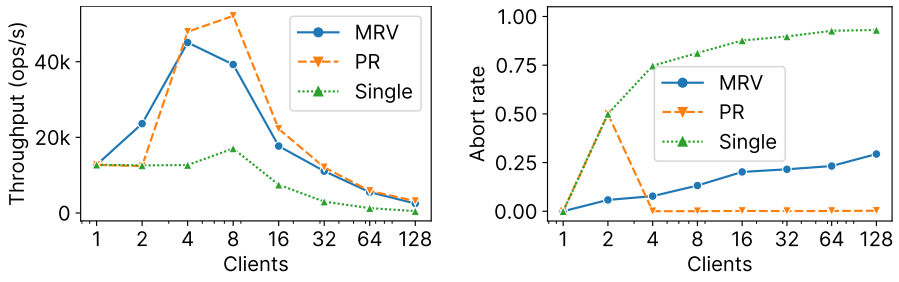
are not able to concurrently update it, as otherwise we would have data inconsistencies due to read/write conflicts. This is made clear in Fig. 4b, where the *single* abort rate increases as the number of clients increases. Both value-splitting techniques significantly reduce the abort rate, with PR nearing a 0% rate due to its split phase. It should be noted that for two clients, we observe a similar behaviour between *single* and PR, since the PR object stays in the joined phase.

It is worth noting that each one of our Kunpeng CPUs houses two NUMA nodes of 32 cores each, which makes the extra memory latency of using more than 32 clients unsuitable for our kind of testing. Even so, our largest throughput is achieved with only eight clients. Thus, this will be our target for the remaining tests.

The padding we mentioned is a simple loop of successive additions to a local variable not used elsewhere. We have used a value of 100K iterations on the previous test, which we found to be the optimal amount as shown in Fig. 5. It is clear that value-splitting is useful independently of the transaction length, but even more so on long-running ones. Beyond the 100K mark, the TM system shows its limitations and the throughput of all the implementations drops significantly. The abort rates on all of them remain consistent, being high on the *single* implementation and low on the value-splitting versions.

The following test measures throughput with a mixed workload of read and write operations, at varying read percentages with eight clients. The results for write and read throughput are shown in Fig. 6.

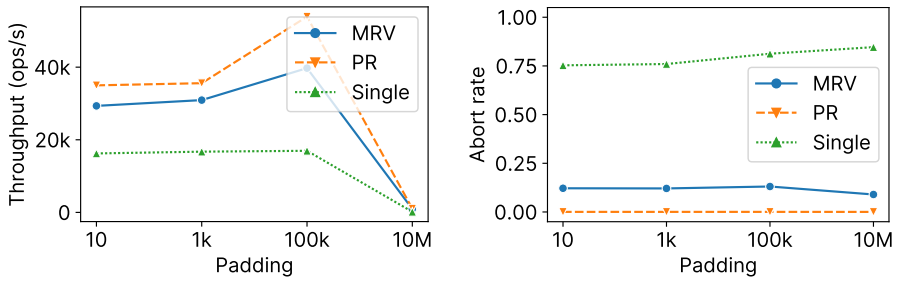
As expected from the first test, in pure write workloads, both value-splitting techniques have a major performance improvement over the baseline code, in terms of write throughput (Fig. 6a). However, as reads are introduced into the workload, the write performance drops by a significant amount, with PR falling below the baseline, as frequently switches back and forth between joined and split phases. With a share of at least 60% read operations, it stays in the joined phase and effectively defaults to a single value. MRVs still mitigate conflicts even with a mix of read and write operations, being able to perform better or on par with the *single* baseline case.



(a) Write throughput.

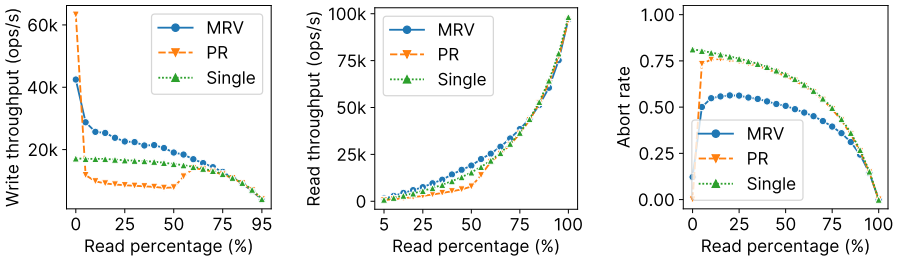
(b) Abort rate.

Fig. 4: Pure write workload with variable number of clients.



(a) Write throughput. (b) Abort rate.

Fig. 5: Pure write workload for eight clients with a variable amount of padding.



(a) Write throughput. (b) Read throughput. (c) Combined abort rate.

Fig. 6: Mixed workload with eight clients.

In terms of read throughput (Fig. 6b), we can observe a similar pattern for the MRVs. They are able to perform slightly better than the *single* strategy in low/medium read workloads, despite having multiple chunks to read. PR again suffers from a performance penalty in low read workloads, which we attribute to the need to wait for the joined phase to proceed with the read.

Looking at the abort rate line plot (Fig. 6c), it is clear that the advantage that PR had on our previous write-exclusive scenarios has disappeared. It is able to achieve a 0% abort rate with a 0% read percentage, as expected, but it quickly follows the abort rate for the *single* value on higher percentages. MRV is able to keep a markedly lower abort rate in write-dominated scenarios, eventually following the other two implementations in high read workloads.

In the end, MRVs are the all-around better option of the three implementations we presented, notably on write-dominated scenarios, which were our initial evaluation target. On these workloads, MRVs offer the best performance by a noticeable margin, while also offering a close-to-baseline performance on read-heavy workloads. It should be noted that, for the latter, the baseline scenario is the target since it already offers the best performance possible, i.e., reading from a single value that is infrequently updated with low contention is better than reading from a value split into multiple chunks, however many they should be.

5 Related work

Performance has been a key issue in TM systems, and there have been several papers that have aimed to analyse its performance over the years. In addition to underlining the relevance of optimisations such as MRVs and PR, these tools could be used to help deploy them. *Syncchar* [17] is a tool for performance prediction and tuning, capable of revealing which parts of the code are more likely to become bottlenecks. Castro et al. [3] propose a non-intrusive solution that intercepts transactional calls in order to collect and trace abort and commit data. Lourenço et al. [12] also propose a TM-specific monitoring framework, in which TM libraries use the provided API to insert the tracking system in their code.

Although we have focused on MRVs and PR, there are other techniques proposed specifically for TM. *Delayed actions* [4], as the name suggests, delay the execution of certain actions until the transactions commit, where they can be executed sequentially to avoid unnecessary aborts. However, this can only be applied to operations whose output is not read in the transactions where they are executed.

Other research has explored ways to reduce read/write conflicts by leveraging object semantics. *Transactional boosting* [7] requires the specification of the inverse operations and the rules for commutativity, allowing the reversal of operations and concurrent transactions to run without conflict, respectively. *Software transactional objects* (STO) [9] leave up to the object the management of its modifications, locking, and version control, while the STO system only acts with abstract reads and writes on these data types. Since these techniques target the semantic meaning of an object as a whole, they could theoretically be combined with value-splitting, e.g., allowing for the concurrent increment of the same MRV chunk without conflicting, which would then avoid the need in our implementation to have as many chunks as it currently does. The multiple chunks would still be useful for conflicting operations.

In addition to MRVs and PR, other techniques have been proposed to handle transactional conflicts in the database space, with various levels of granularity. Just like MRVs and PR, Escrow Locking [16] and various Distributed Reservations techniques [2,1,11,13] also target conflicts on numeric fields while ensuring lower bound invariants. Both of these solutions rely on reserving a private amount of the total value to avoid/reduce concurrency-induced conflicts. In Escrow Locking, the reservation is made at the start of a transaction to a central entity, e.g., a latch, while in Distributed Reservations the amounts are preallocated to each site in a database cluster. Cassandra counters [21] also split a value across multiple shards, but do not support lower-bound invariants. Conflict-free Replicated Data Types [19] also target conflicts in the same field, by allowing concurrent transactions to update in parallel and eventually merging the results. With lower granularity, Timestamp Splitting [10] is designed to reduce conflicts in accesses to different fields of the same record, by assigning different timestamps to different column subsets. Finally, some systems handle transactional conflicts by committing independently and later completely discarding some modifications, using weaker consistency rules such as Last-Writer-Wins [20].

6 Conclusions

Preliminary results show that value-splitting is worth exploring in the context of transactional memory systems. As demonstrated, PR helps in extreme workloads, offering the best performance where at any given moment either only writes or reads are executed. MRVs, on the other hand, show that their biggest strength lies in their adaptability, achieving great performance in write-exclusive scenarios, the best read and write performance in write-heavy workloads, and on-par performance with the single value for read-dominated situations. The analysis of these types of write-heavy workloads is of relevance, since they can be easily found in the online sale of highly contended items, e.g., limited-edition releases or concert ticket sales for popular artists.

In future work, we believe that the adjustment of the value-splitting parameters and the usage of different underlying structures could have a significant impact on the performance achieved. Workloads with dynamic amounts of contention over time could also prove useful in emphasising the advantage of MRVs over static thread assignment of PR.

In this paper, we have only focused on simple integer values for our splitting techniques, but we could also apply some of our insights to more complex data structures. The original PR proposal is already applicable to ordered tuples and top-K sets, which could be an interesting addition to value-splitting in TM.

There already exist reference benchmarks for TM, such as the STAMP [14] suite – which also heavily focuses on updates over numerical values – but they are not adapted to work with object-based systems, as is the case of Wyatt-STM. Future work could include the port of one or more of the benchmark applications to Wyatt-STM, to better measure the applicability of value-splitting to more realistic workloads.

References

1. Balegas, V., Serra, D., Duarte, S., Ferreira, C., Shapiro, M., Rodrigues, R., Preguiça, N.: Extending eventually consistent cloud databases for enforcing numeric invariants. In: IEEE 34th Intl. Symp. on Reliable Distributed Systems (SRDS). pp. 31–36 (Sep 2015). https://doi.org/10.1109/SRDS.2015.32
2. Barbará-Millá, D., Garcia-Molina, H.: The demarcation protocol: A technique for maintaining constraints in distributed database systems. The VLDB Journal **3**(3) (1994). https://doi.org/10.1007/BF01232643, https://doi.org/10.1007/BF01232643
3. Castro, M., Georgiev, K., Marangozova-Martin, V., Méhaut, J.F., Fernandes, L.G., Santana, M.: Analysis and tracing of applications based on software transactional memory on multicore architectures. In: 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing. pp. 199–206 (2011). https://doi.org/10.1109/PDP.2011.27
4. Diegues, N., Romano, P.: Bumper: Sheltering distributed transactions from conflicts. Future Generation Computer Systems **51**, 20–35 (2015). https://doi.org/https://doi.org/10.1016/j.future.2015.04.002,

https://www.sciencedirect.com/science/article/pii/S0167739X15000941, special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems
5. Faria, N., Pereira, J.: MRVs: Enforcing numeric invariants in parallel updates to hotspots with randomized splitting. Proc. ACM Manag. Data (SIGMOD) **1**(1) (2023). https://doi.org/10.1145/3588723
6. Hall, B.: Wyatt-STM (May 2023), https://github.com/bretthall/Wyatt-STM, original-date: 2015-12-18T17:34:53Z
7. Herlihy, M., Koskinen, E.: Transactional boosting: a methodology for highly-concurrent transactional objects. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 207–216. PPoPP '08, Association for Computing Machinery, New York, NY, USA (Feb 2008). https://doi.org/10.1145/1345206.1345237
8. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2), 289–300 (may 1993). https://doi.org/10.1145/173682.165164, https://doi.org/10.1145/173682.165164
9. Herman, N., Inala, J.P., Huang, Y., Tsai, L., Kohler, E., Liskov, B., Shrira, L.: Type-aware transactions for faster concurrent code. In: Proceedings of the Eleventh European Conference on Computer Systems. pp. 1–16. EuroSys '16, Association for Computing Machinery, New York, NY, USA (Apr 2016). https://doi.org/10.1145/2901318.2901348
10. Huang, Y., Qian, W., Kohler, E., Liskov, B., Shrira, L.: Opportunities for optimism in contended main-memory multicore transactions. Proc. VLDB Endowment **13**(5), 629–642 (2020)
11. Liu, J., Magrino, T., Arden, O., George, M., Myers, A.: Warranties for faster strong consistency. In: 11th USENIX Symp. on Networked Systems Design and Implementation (NSDI 14). pp. 503–517 (2014)
12. Lourenço, J.a., Dias, R., Luís, J.a., Rebelo, M., Pessanha, V.: Understanding the behavior of transactional memory applications. In: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. PADTAD '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1639622.1639625, https://doi.org/10.1145/1639622.1639625
13. Magrino, T., Liu, J., Foster, N., Gehrke, J., Myers, A.: Efficient, consistent distributed computation with predictive treaties. In: Proc. 14th EuroSys Conf. 2019. pp. 1–16 (2019)
14. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: 2008 IEEE International Symposium on Workload Characterization (Sep 2008). https://doi.org/10.1109/IISWC.2008.4636089
15. Narula, N., Cutler, C., Kohler, E., Morris, R.: Phase reconciliation for contended in-memory transactions. In: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation. pp. 511–524. OSDI'14, USENIX Association, USA (Oct 2014), https://dl.acm.org/doi/10.5555/2685048.2685088
16. O'Neil, P.E.: The escrow transactional method. ACM Trans. on Database Systems (TODS) **11**(4), 405–430 (1986)
17. Porter, D.E., Witchel, E.: Understanding transactional memory performance. In: 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). pp. 97–108 (2010). https://doi.org/10.1109/ISPASS.2010.5452061
18. Puente, J.P.B.: Persistence for the masses: Rrb-vectors in a systems language. Proceedings of the ACM on Programming Languages **1**(ICFP), 1–28 (2017)

19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Symp. on Self-Stabilizing Systems. pp. 386–400. Springer (2011)
20. Thomas, R.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. on Database Systems (TODS) **4**(2), 180–209 (1979)
21. Yeschenko, A.: The DataStax blog - what's new in Cassandra 2.1: Better implementation of counters. https://www.datastax.com/blog/whats-new-cassandra-21-better-implementation-counters (2014)