Paper report for CS240 (Computer Systems and Concurrency), King Abdullah University of Science and technology, Fall 2014
Instructed By: Prof. Hany Ramadan

Paper Report on Phase Reconciliation for Contended In-Memory Transactions

Uchenna Akujuobi
Uchenna.akujuobi@kaust.edu.sa

Abstract

In this paper report, we examine phase reconciliation works and how it solves the problems of contention based on the paper *Phase Reconciliation for Contended In-Memory Transactions* by *Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris*. The authors also presented *Doppel,* a phase reconciliation database. Due to today's advancement in technology, there has been a rapid increase in multiprocessor systems and this has led to the increased need to develop concurrent systems that can make efficient use of all the available resources. Concurrent systems has been used to build applications scale these processors. While the number of cores per chip grows exponentially, the problem of how to make effective use of these resources has not been fully solved and cannot be fully ignored.

INTRODUCTION

To gain more performance and make maximum use of the cores, a process should execute in parallel which should result in increased speed in process execution, resources used to the maximum capacity, better performance, etc. This would have been perfect if it had been the case. But, there are problems that arises that would not have been there if all processes were executed serially. As we know, processes executing in parallel also need access to some resources, communicate with each other and protect invariants and this can be troublesome provided that two or more processes might need to access or make changes to a particular resources at the same time and will try at the same time to gain access to these resources and the result of this action can never be good. This is not a new problem in computer science and have been different people have come up with ways to try to addressed this issue.

This paper is focused on the contentions that happen in memory by transactions. This particular issue has been addressed by Database concurrency control protocols (2PL and OOC). In 2PL (two phase locking), a transaction that needs a resource being used by another resource will wait for the other transaction by just spinning on a lock while in OOC (optimistic concurrency control), the transaction will wait for the other by aborting and trying again to see if the resource is available. Both methods force the transactions to execute serially which defeats the concept of parallel execution.

Unfortunately, this problem is something we cannot escape from in the real world. For instance, lets consider an IPhone 6 listed on eBay to be sold as on bid or to the best offer with a lot of watchers (people biding or monitoring the item to bid at the last minute). Getting to the end time of the item, a lot of people will be sending there offers and it can happen that many people send an offer at the same time, modern multicore databases will execute these actions serially and might not get to the real highest offer before the time ends thereby, giving a wrong amount. So this item will end up being sold not to the highest bidder but to the last offer the database was able to update before the time ran out.

This paper presents a new concurrency control technique *phase reconciliation* which can execute some highly conflicting workload efficiently in parallel and still guaranties serializability; and also presents *Doppel* which is a phase recognition based In-memory database. Phase reconciliation is based on shifting data between splits and reconciled phases dynamically. Phase transactions can be executed either in joint phases or in split phases. In joined phases, there are no splits and therefore, no per core values. So, here database structures are accessed using OCC. In split phases, to reduce contention, data is split and operations are done in different cores. These different cores make updates to their per-core values and which after executions on each core; the results are reconciled to a global store by short reconciliation phases. When a reconciliation phase ends, blocked transactions resume.

Work has been done to try to address issues that arise on transactional memory, database concurrency control and Distributed consistency. Phase reconciliation got inspired by most of these works and adopted some of them in its design. In this paper I would be reviewing the Phase Reconciliation for contended In-Memory Transactions paper [1] and also will be discussing the techniques used in Dora [2] in implementing the Main-memory database concurrency control, the design of Sync-Phase [3] transactional memory and how RedBlue consistency [4] is applied to keep the distributed consistency property. In this paper, I would look into the design and implementation of phase reconciliation in the context of Doppel and also into these other 3 techniques.Now we will discuss of the implementations.

DORA

DORA is a data oriented architecture that tries to solve the problem of using locks which generally impedes performance. Pandis, Johnson, Hardavellas, and Ailamaki in the paper Data Oriented Transaction execution noted that the main cause of cause of contention is the traditional tread-to-transaction assignment policy. On the conventional transaction where each thread runs a separate transaction, access to data by the transactions during a shared data access leads to contention.

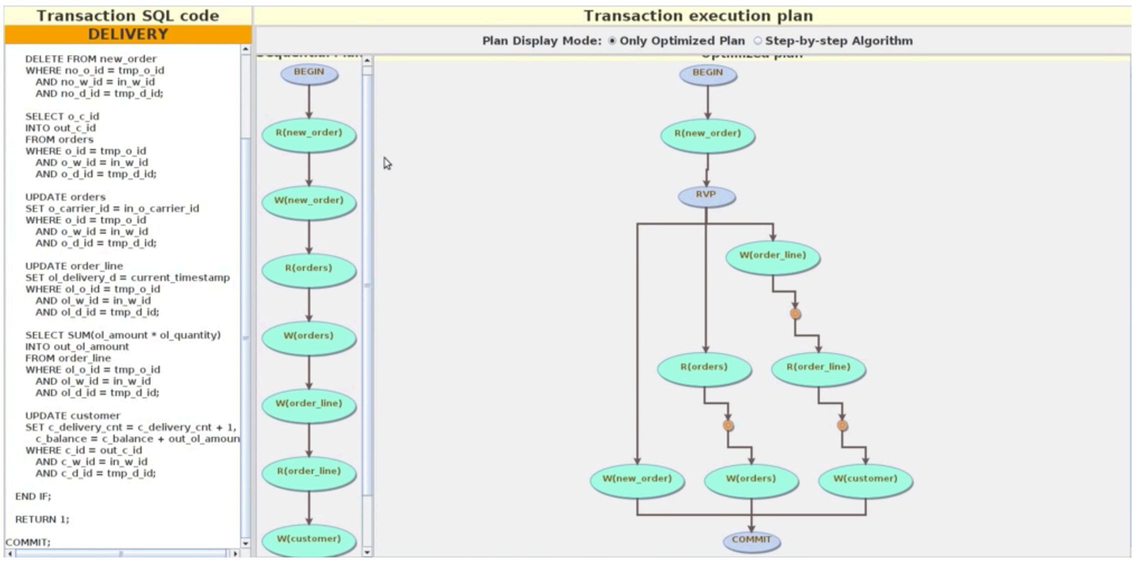
The access patterns here are uncoordinated and so to ensure integrity, different lock mechanisms are used. These lock mechanisms however, creates overhead and reduces performance. Experiments conducted by Pandis, Johnson, Hardavellas, and Ailamaki showed that the system spends more than 85% of its execution time on threads waiting to critical sections in side the lock manager. A new data oriented achetecture OLTP was proposed by the authors that could drastically reduce the contention problems. And this was evaluated using a prototype DORA.

This system, rather than each thread being coupled with a transaction, threads in DORA has its own subset of database. DORA uses a mechanism the authors called thread-to-data assignment where transactions are broken down into smaller actions and are routed to different threads of execution depending on the data required by the transactions. DORA handles the distribution of computations while each thread coordinates access to data in its subset. This limits the interaction between threads and removes some of the contention problems. The routing /flow of transactions from one thread to another as they assess different parts of the database is done with minimal overhead as DORA makes use of multicore systems high-bandwidth, low-latency inter-core communications.

The main functionalities of DORA are binding to worker threads to subset of the database, work distribution of each transaction to the transaction executing threads according to the data accessed by the transaction and last functionality is interaction avoidance during request executions using centralized lock manager.
Database binding to worker threads is done by setting a routing rule for each database table. Worker threads are called executors and each dataset is assigned to an executor. Multiple dataset from a single table can be assigned to one executor. The routing rule according to the authors is a mapping of sets of records, or datasets, to worker executors. Routing rule has no requirement, which is that each possible record of the table need to map to a unique dataset. Routing rules changes periodically at runtime to balance load and are managed and updated by DORA. During transaction execution, different actions are made to the database. An action is a subset of a transaction code that requires access to the database. This access can be to a single record or small sets of records from a single table. A graph of these actions to the database is known as a *transaction flow*. For DORA to distribute/route transactions from one executor to another, it needs to know the action and subsets of database which the action that a transaction executes. This it does by translating each transaction to a transaction flow graph. Each action has an identifier, which is used to identify the records the action needs to access. This identifier can be a set of values for the routing field (routing field can any combination of the columns of the table) or an empty space (actions without any routing field) depending on the type of access required. The more specific an action identifier is, the more easily the transaction involved with that action could be routed. Two actions however can be merged if they have the same identifier.

In order to control distributed execution of a transaction and transfer data between actions that have data dependencies, DORA uses rendezvous points or RVP. RVP or rendezvous points are the shared objects across actions of the same transactions. We know that it can be possible for more than one action to require access to the same data and when this happens, it can lead to bottlenecks even in DORA. DORA addresses this issue by using rendezvous points. Whenever two actions are data dependent, execution of a transaction are separated to different phases by rendezvous points. A counter was added to each RVP that is initially set to the number of actions that need to report to it. This counter is decremented by every executor that has finished executing an action and when the counter becomes zero, the last executor that zeroed the counter starts the next phase. Figure 1 shows a demo on how a transaction code (SQL), which would usually be serially executed, is optimized in by DORA.

DORA moves all actions that makes use of the same dataset to one executor. Three data structure are associated with each executor and they are an incoming actions queue, a completed action queue and a thread-local lock table. The incoming queue contains incoming actions and these actions are executed according to the order they enter the queue. The executor maintains isolation and ordering across conflicting actions. And for this, it uses the local lock table to detect conflicting actions and the conflict resolution happens at the action identifier level. The local locks have a shared mode and exclusive mode and an action needs to acquire the local lock before proceeding. When the lock is obtained, it executes without a centralized concurrency control. As for load balancing, the load of each executor is monitored by the Dora resource manager monitor and actions are taken if the average load an executor is assigned is a lot larger than the rest. This action taken is usually the resizing the data assigned to each executor. This action however, is not free.



```
  DELETE FROM new_order
  WHERE no_o_id = tmp_o_id
    AND no_w_id = in_w_id
    AND no_d_id = tmp_d_id;

  SELECT o_c_id
  INTO out_c_id
  FROM orders
  WHERE o_id = tmp_o_id
    AND o_w_id = in_w_id
    AND o_d_id = tmp_d_id;

  UPDATE orders
  SET o_carrier_id = in_o_carrier_id
  WHERE o_id = tmp_o_id
    AND o_w_id = in_w_id
    AND o_d_id = tmp_d_id;

  UPDATE order_line
  SET ol_delivery_d = current_timestamp
  WHERE ol_o_id = tmp_o_id
    AND ol_w_id = in_w_id
    AND ol_d_id = tmp_d_id;

  SELECT SUM(ol_amount * ol_quantity)
  INTO out_ol_amount
  FROM order_line
  WHERE ol_o_id = tmp_o_id
    AND ol_w_id = in_w_id
    AND ol_d_id = tmp_d_id;

  UPDATE customer
  SET c_delivery_cnt = c_delivery_cnt + 1,
    c_balance = c_balance + out_ol_amoun
  WHERE c_id = out_c_id
    AND c_w_id = in_w_id
    AND c_d_id = tmp_d_id;

 END IF;

 RETURN 1;

COMMIT;
```

Figure 1. Graphical representation of DORA’s execution plan using Shore-MT

SYNCHRONIZATION PHASES

As transactions read and write to memory concurrently, the transactional memory system needs to check for modifications and changes to a memory region between two memory accesses in other to maintain the consistency. This validation is not cheap because, the read set usually increases as the transaction progresses. The *read set* is the list of memory locations needed to be checked.

J. Schneider, F. Landau, and R. Wattenhofer however, took another approach to address the problem of concurrency in transactional memory. This approach is however different from DORA in that it was based on Thread-to-transaction policy. They introduced a phase-based technique where time is divided into two different phases: the computational phases and commit phases (the synchronization phase). This technique is called *Sync-phase technique*. Here, active transactions do computations during the computation phase and are allowed committing only during the commit phase. Figure 2 shows a transaction read set validation in ordinary DSTM and Sync-Phase DSTM.

Phases can be determined through time or by some other means. During the commit phase, transactions that have something to commit, commit the transaction and those with nothing to commit remain idle during this phase and have to validate their read set and states after each commit phase.

J. Schneider, F. Landau, and R. Wattenhofer proposed a technique of which the basic idea is to execute multiple commits at the same time to make them look like one single commit to an outside observer. The Sync-phase technique alternates commit (synchronization) and the computation phases. Transactions that finish after a previous commit phase have to wait until the next commit phase. The duration of each phase can be set according to preference. This can be done either using an algorithm that sets the phase duration at runtime or by using fixed time interval. As computations take longer time, the duration of the computation phase needs to be allocated a longer time but not too much as the transaction which finished their computation and needs to commit will have to wait a long time before committing. A middle ground has to be found in other to get a good performance. J. Schneider, F. Landau, and R. Wattenhofer got rid of the explicit use of the phase duration in Sync-phase STM (Software transactional memory) in order to increase performance. There are two rules to start and end commit phases:

1. A commit phase starts if a fraction of all transactions is ready to commit: $N_c/N \geq c_0$ for $c_0 \in [0, 1]$
2. A commit phase ends if a fraction of all transactions has committed: $N_c/N \leq c_0$ for $0 \leq c_1 < c_0$

where N_c is the number of transactions ready to commit, N is the total number of active transactions (but temporary sleeping).

Rule 1 prevents time wasted by waiting for the next commit phase if enough transactions can commit. Ideally, the choice of c_0 is made such that the cost of waiting of the N_c transactions balances the costs of validations of the other N - N_c transactions.

Rule 2 tries to handle the case where only few transactions wants to commit which would be better to stop the commit phase but also allow the

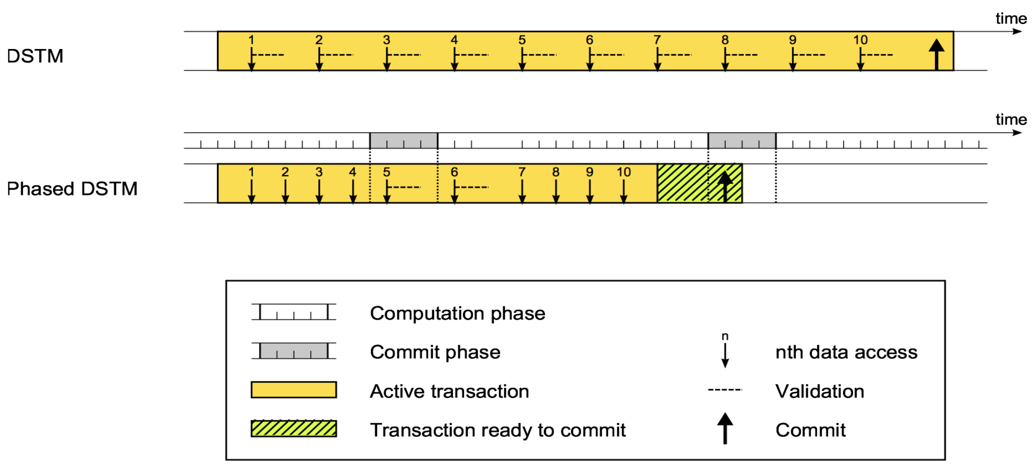


Figure 2. Validation of read sets in DSTM and Sync-Phased DSTM

transaction which are ready to commit to commit

RadixVM

The problem of memory contention is not only related to transactional memory. Multithreaded applications, which run on many core processors, also face the same problem. Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich took another turn in looking into the problem of multithread memory contention not just using software transactional memory but using virtual memory systems. Using this approach, they considered multithreaded applications instead of transactions.

They focused on ways to make memory allocation, freeing and page fault using mmap and mumap calls which applications use to allocate and return memory to the operating system perfectly parallelizable for threads of a process that use non-overlapping regions in the shared address space. Serializing mmap and mumap calls applications can easily be bottlenecked by contentions in the operating system. RadixVM is a virtual memory system designed to enable fully concurrent operations on shared address space for multithread processes on cache-coherent multicore computers in which virtual memory system operations contend only if they operate on overlapping memory regions.

RadixVM uses three different ideas to enable mmap, mumap and pagefault to scale perfectly in non-overlapping memory regions. First, it uses radix tree to record mapped memory regions. Second, it uses a scalable reference counter to track the sates of physical pages to know if they are free and no radix tree nodes are used or not if they are not free. Third, it avoids shooting down hardware TLBs that don’t have the page mapping cached whenever a page must be unmapped. RadixVM itself was implemented in a research kernel derived from xv6. One of the main contributions of RadixVM is reducing the number of cores that must be contacted to do a TLB shoot down. This was done by using a scheme that tracks which cores may have each page mapping cached to their TLBs, thus allow RdixVM to send TLB shootdown interrupts only to those cores, and remove TLB shootdowns for mappings not shared between cores. RadixVM uses a radix-tree-based data structure, which allows for concurrent non-overlapping lookups. Deletion, and inserts without using software transactional memory. In RadixVM, if k number of threads in the same process deallocates memory, then the mumap scale perfectly so, they will take the same amount of time to execute and does not depend on k. In the same way, if one thread executes mmap and another thread executes the mumap, they won't slow each other down provided they do not use overlapping memory regions. If they use overlapping memory regions, then they would be serialized. RadixVM uses Refcache a simple reference counting scheme to keep track and retake memory pages and radix nodes. For instance, to free underlying physical pages, it has to make sure that there is no other thread using the same physical pages and thus needs to make use of a reference count to know when there are threads using the physical pages. Let Nrc be the number of reference counted objects and Nc be the number of cores, Refcache requires space proportional to Nrc + Nc which is less than other scalable reference counting mechanisms which require Nrc x Nc. It implements a per-core reference delta caches and is targeted at workloads that can tolerate some latency in reclaiming resources and when increment and decrement occur in the same core. For instance, the same thread that faulted pages into a mapped memory region also unmaps it.

In Refcache, each reference counted object has a global count and each core maintains a local, fixed cache of deltas to objects' reference counts. Incrementing and Decrementing an object's reference count only modifies the local cached delta that is periodically flushed to the objects global reference count. The true count can be gotten by the summing an object's global count and the local delta for an object found in the per-core delta caches. Refcache divides time into periodic epochs (in which each core flushes all the reference count deltas in its local cache, applying updates to the global reference count of each object), which it uses to determine a zero true count. Once the true count drops to zero, there would be no updates and thus when the global reference count drops to zero and remains zero for an entire epoch, it can be guarantied that the true count is zero and the object is freed.

To support the tracking of objects that may have been deleted Refcache uses weak references, which provide tryget operation that will either increment the object's reference count and return the object, or will indicate that the object has been deleted. The weak reference is a pointer with a dying bit (set when the object's global reference count first drops to zero) and a back-reference from the referenced object. When Refcache decides to free the object, it first automatically clears the both the dying bit and the pointer in the weak reference and then frees the object but if this doesn't succeed, it reexamines the object after 2 epochs later.

PHASE RECONCILIATION

Design

Phase reconciliation design is based o research in 4 different areas: transactional memory, main-memory database concurrency control, multicore scalability, and Distributed consistency. Phase reconciliation makes contributions in each of these areas.

In transactional memory, since transactions are often very large to use hardware transactional memory, Phase memory developed techniques to split transactions and apply them using time stamp ordering. This technique helps in situations where spurious aborts are common. Unlike Sync-Phase, it doesn't split transactions into compute and commit phases and unlike DORA, it doesn't partition data and running one partition per core but partitions local copies of the data amongst cores for read and write and provides a way to remerge the data for access by other cores. In Main-memory database concurrency control, by restricting transaction execution to phases, phase reconciliation makes it possible for transactions to commit without global communications.

In multicore scalability, Phase reconciliation aimed to reduce the burden shifted onto reads by other methods by amortizing the effect of reconciliation over many transactions and making these ideas work for bigger transaction systems.

In distributed consistency, by restricting operations only during phases but not between them, phase reconciliation supports both scalable implementations of commutative operations and efficient implementations of non-commutative operations on the same data items.

Phase reconciliation was implemented in a multicore, in-memory database called Doppel. Transactions in Doppel once begun run to completion without communication or disk Input/output and this means that transactions would not be affected or blocked by disk stalls or user stalls. Doppel also uses the concept of worker threads just like in DORA. These worker threads are one per core, run transactions. Doppel records have types and transactions interact with the database using calls to operations. There are different operations in Doppel. Some of these operations return values, some don't return values. Some operations modify the database and some don't and each database operations access just one database record but users can build multi record operations from single-record ones using transactions. For instance, the G_{ET}(k), returns the key k and doesn't modify the database.
During split and reconciliation phase, Doppel marks contended database records as splits. For these records, operations that would normally contend would proceed in parallel.

- At the beginning of each split phase, Doppel initializes per-core slices for each split record. There is one slice per contended record per core.
- During the split phase, all operations on split records are applied to their per-core slices.
- During the reconciliation phase, the per-core slices are merged back into the global store.

Figure 3 shows an example of a split record with per core slices in two cores.

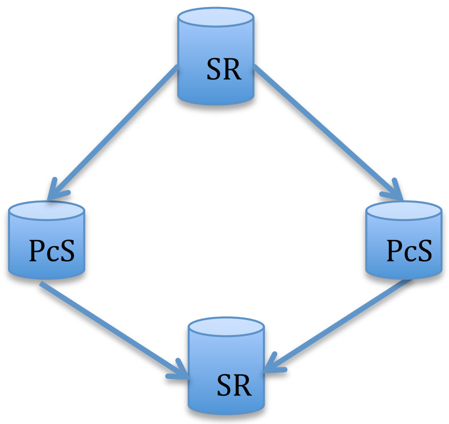


Fig 3. SR- Split record, PcS- Per core slice

To ensure good performance, per-core slices must be quick to initialize, and operations on slices must be fast. To ensure efficiency, the combination of applying the operation to a slice and the merging of the slices should have the same effects as the operation would normally in some serial order. To ensure correctness, serializability must be ensured in Doppel. However, the code required to update a slice may be different from the code needed to update a normal record. There must exist a serial order of transactions that split which satisfies:

- The result of merging per-core slices with the global store is the same as if the transactions had executed, in the serial order, against the global store.
- Every operation executed on a split record gets the same return value as if it had executed, in the serial order, against the global store.
- Every operation executed on the global store gets the same return value as it would in the serial order.

Doppel supports several splittable operations and to ensure these operations are correct and fast, splittable operations have to follow the following guidelines:

- Every splittable operation must commute with itself.
- Every splittable operation must return nothing.
- The system selects one splittable operation per split record per split phase. The selected operation can change between phases—for example, the operation for key k might be M in in one split phase, and Max in the next—but within a given phase, any operation but the selected operation causes the containing transaction to abort (and retry in the next joined phase).
- The size of a per-core slice is independent of the number of operations that executed on that slice.

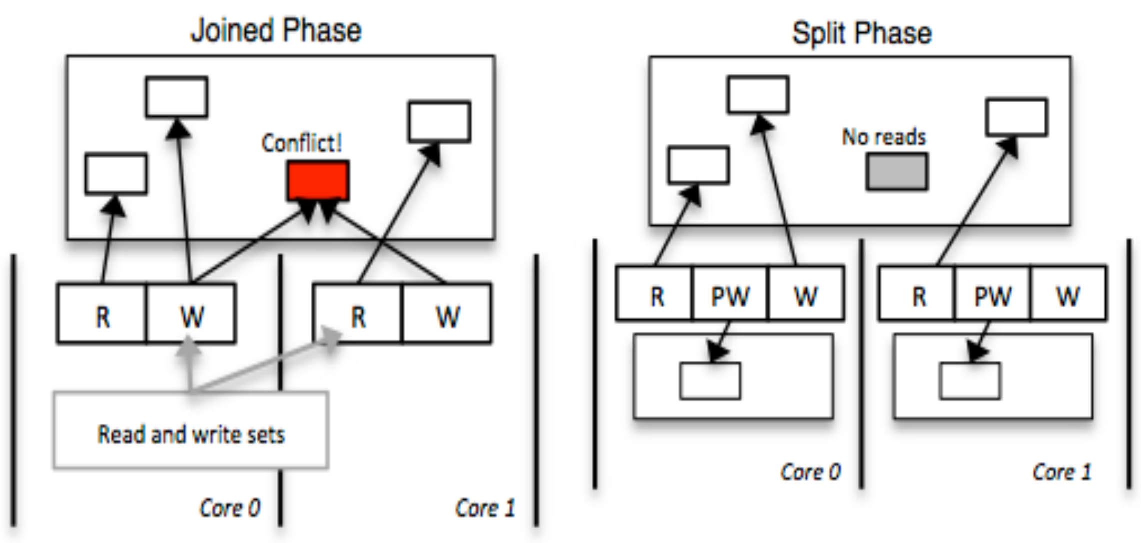
The splittable operations in Doppel are:

- M_{AX}(k,n), M_{IN}(k,n) – Replaces the integer k with the maximum or minimum of it and n.

- A_{DD}(k,n) – Adds n to the integer k.
- OPUT(k,o,x) – Operation on ordered tuples.
- $T_{OP}KI_{NSERT}$(k,o,x) – Operation on top-K sets.

More operations could also be easily added.
To look into how Doppel executes splittable operations, lets take for instance M_{IN}(k,n) which replaces k with the minimum of k and n and returns nothing.
When Doppel detects contention on M_{IN}(k,n) operations for some key k, it marks k as split for M_{IN}. When entering the next split phase, Doppel initializes per-core slices c_j[k] with the global value v[k]. When a transaction on core j commits an operation M_{IN}(k,n), Doppel sets cj[k]$\leftarrow$ M_{IN}\{cj[k],n\}. Key k is temporarily reserved for M_{IN} operations; a transaction that tries to execute another kind of operation on k will block until the following joined phase. When the split phase is over, Doppel merges the per-core slices by setting v[k]$\leftarrow$ cj[k]$\leftarrow$ $\min_j c_j$[k]. If many concurrent transactions call M_{IN}(k,n) during a split phase, Doppel executes them in parallel on multiple cores with no coordination, getting good parallel speedup over the serial execution of conventional OCC or locking.

Joined phase execution
Any transaction can be executed in a joined phase. There is no notion of split data and no per-core slices so, the protocol treats all records the same. Thus joined phase can use any concurrency control protocol but if everything work as planned, the joined phase will have fewer conflicts since transactions that conflicts should run in split phase, it would make more sense to use a protocol that would perform well when conflicts are rare. This is why Doppel's joined phase uses optimistic concurrency control. Each transaction executes within a single phase. Any transaction that commits in a joined phase executed completely within that joined phase. Doppel thus cannot leave a joined phase for the following split phase until all current transactions commit or abort. Fig 4a shows two transactions running in joined mode.



(a) (b)

Figure 4. (a) Concurrent transactions executing on different cores in joined phase. (b) Concurrent transactions executing on different cores in split phase.

Split phase execution

Some transactions that would normally conflict can be executed in parallel in split phase. Split-data operations execute on per-core slices. A transaction that invokes an unselected operation on a split record will be aborted and stashed for restart during the next joined phase. Doppel doesn't need to lock slices or check their version numbers since the slices are invisible to concurrently running transactions. Just as in joined phase, any transaction that commits in split phase executed completely within that same split phase; Doppel does not enter the next joined phase except all of the split-phase transactions commit or abort. Fig 4b shows a split phase, with each transaction writing to per-core slices.

Reconciliation phase execution

This phase comes after a split phase. During this phase, each core stops processing transactions and merges its per-core slices with the global store. This involves serial processing of the per-core slices since each core has to lock the global record, updates the value and the releases it. But the expense of serial processing is amortized over all the transactions that executed in the split mode. The per-core slices are then cleared and the database enters the next joined phase.

Coordinator threads manage transitions between phases. Phase transitions occurs in 3 steps:

1. Coordinator publishes the phase change in a global variable.
2. All Workers check this variable and if a change is noticed, they stop processing transactions in the current phase, acknowledge the change and wait for permission to continue.
3. Coordinator releases the workers and they start executing transactions in alternate phase.

For instance, to initiate a transition from a split phase to the next joined phase, the coordinator publishes the phase change in a global variable. When a split-phase worker notices a transition to the reconciliation phase, it stops processing transactions, merges its per-core slices with the global store, and then acknowledges the phase transition and waits for permission to proceed. Once all workers have acknowledged the change, the coordinator releases them to the next joined phase; each worker restarts any transactions it stashed in the split phase and starts accepting new transactions. It is safe for reconciliation to proceed in parallel with other cores' split-phase transactions since reconciliation modifies the global versions of split records, while split-phase transactions access per-core slices. The Doppel coordinator usually starts a phase change every 20 milliseconds, but feedback mechanisms allow it to flexibly adjust to the workload.

To decide which transaction to move to execute in split phase, Doppel samples transactions conflicting record access during the joined execution and keeps a count of which records are most conflicted and by which operations. During transition to split phase, a coordinator thread examines these counts and mark the most conflicted records as split data for the next phase. Each cores reads this list before the start of the next phase in order to know which records are restricted. And to decide which transaction to move back to execute from split phase to joined phase, Doppel samples which transactions are stashed due to incompatible operations on split data during the split phase. Doppel also supports manual data labeling.

Doppel is implemented as a multithreaded server written in Go and runs one worker thread per core, and also one coordinator thread which is responsible for changing phases and synchronizing workers when progressing to a new phase. Doppel uses channels to synchronize phase changes and acknowledgements between the coordinator and workers. Workers read and write to a shared store, which is a set of key/value maps, using per-key locks. The maps are implemented as hash tables.

Experiments and Discussion

In this section we are going to look into the experiment performed by these different approaches to solving memory contention problem. I would have wanted to compare the throughput of all the approaches side by side but they all based their experiments on different measurements. Thus, in this section we will be discussing some of the experiments performed by these approaches, the good and the shortcomings of the different approaches.

First let us look into DORA which came in 2010. To show the difference between thread-to-transaction and thread-to-data, the performance overhead of critical section contention of DORA and was measured against a thread-to-transaction. It depicts the throughput per CPU utilization attained by a state-of-the-art storage manager as the CPU utilization increases. Transactions from three OLTP benchmarks: Nokia's Network Database Benchmark or TM-1 [19], TPC-C [20], and TPC-B [1] were used. Transaction accesses only 1-4 records, and must execute with low latency even under heavy load. A database of 5M subscribers (~7.5GB) was used. The result of this is shown in Fig 5. The result gotten shows that as the machine utilization increases, the performance per CPU utilization drops for the baseline.

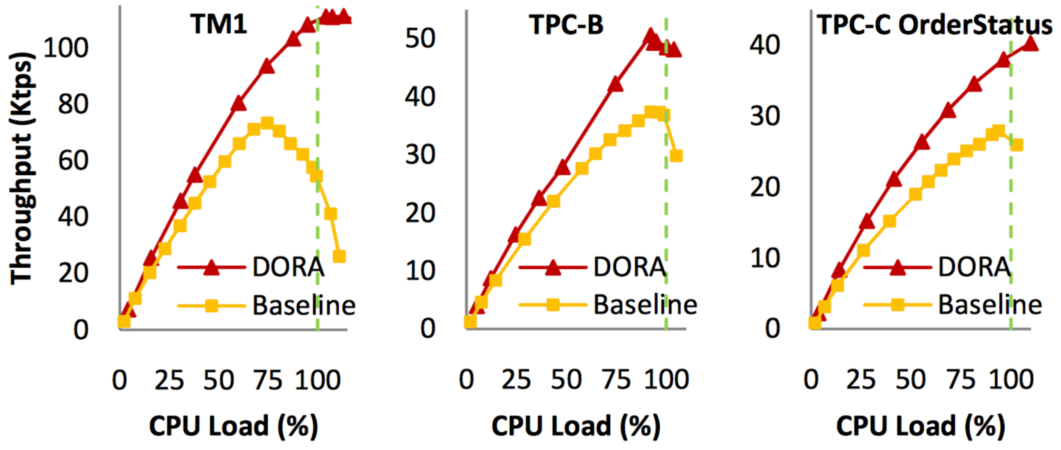
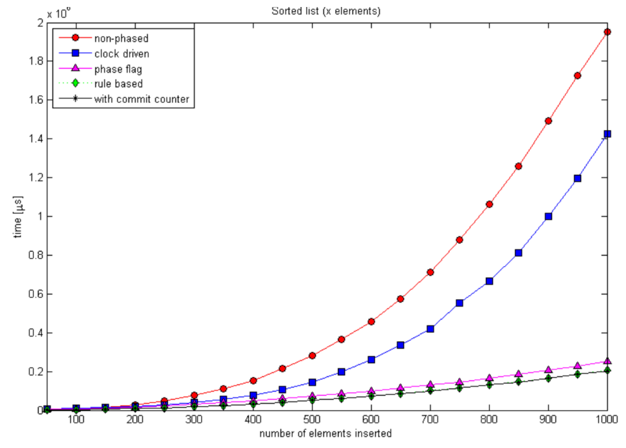


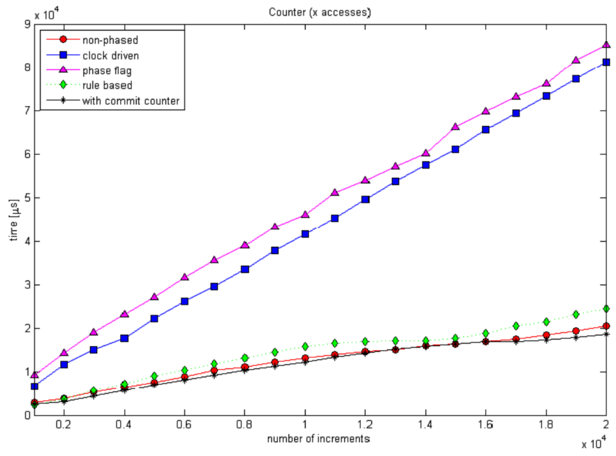
Fig 5. The time breakdown of the performance of in executing transactions from TM-1, TCP-C and TCP-B

For the experiment, a conventional transaction processing system Shore-MT was used as the baseline. I would assume that at that time, this is a state of the art system to be used for a comparison. The bad performance in the baseline can be attributed to the uncoordinated and arbitrary access pattern of each transaction and since transactions run on separate threads, they tend to contend with each other during shared data access. This is one of the problems phase reconciliation tries to solve. The downside is that the over head of cross-partition transactions is significant, and finding a good partition can be difficult.

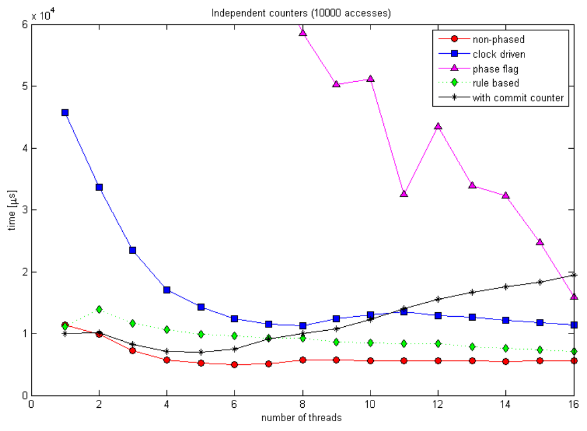
The second approach is synchronization phases, which came later in 2011 with the idea of dividing time into different phases in other to speed up transactional memory. They implemented this idea in three ways: clock driven, phase flag and rule based. The first uses phases of fixed length and is clock driven. The second uses phases that depend on a flag implemented in the software and the third uses a global counter that is incremented when ever a transaction commits. A transaction only needs to validate its read set, given that the commit counter changed since the last validation. These implementations and the non sync-phase DSTM were compared using (a)sorted list, (b)global counter and (c) independent counter benchmarks. The results gotten in Figure 6 shows that non-phased DSTM always performed worse with others varied. It would have been interesting to compare this with DORA to see the improvement in throughput against the thread-to-data systems. The down side of this approach is how to select the optimal duration for each phase.



(a)



(b)



(c)

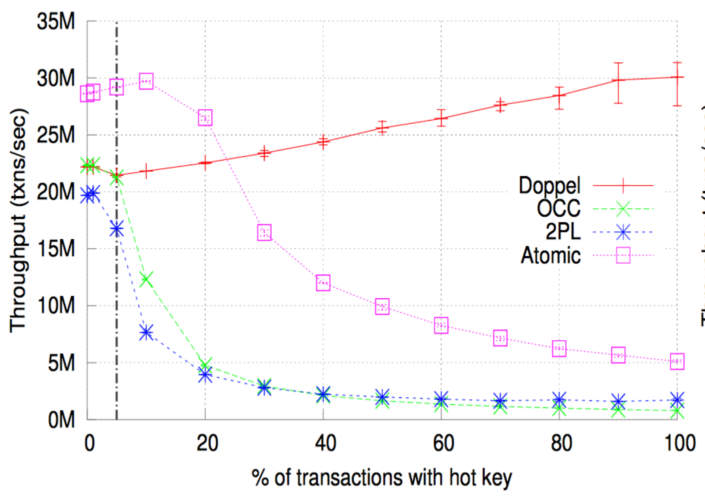
Figure 6. Comparison of the different implementations of DSTM.

The next is RadixVm that came in 2013 and where interested in virtual memory. The goal was to achieve full concurrent operations on shared address spaces for multithreaded processes on cache-coherent multicore computers. They argued that some of the features in software transactional memory was not really explained well to be used in a virtual memory system so they wanted to allow some of these concurrent features for non overlapping inserts, delete and look ups without the need for STM and also to improve the performance of TLB shootdowns by using a radix-tree data structure. Even though they succeeded and there were a lot of experiments that proved that, there was a downside to it. Since radix a tree is less compact than the binary tree representations of virtual memory metadata, the memory overhead increased in RadixVm and this can be attributed also to the fact that page tables are per core instead of shared. Figure 7 shows the difference in memory usage.

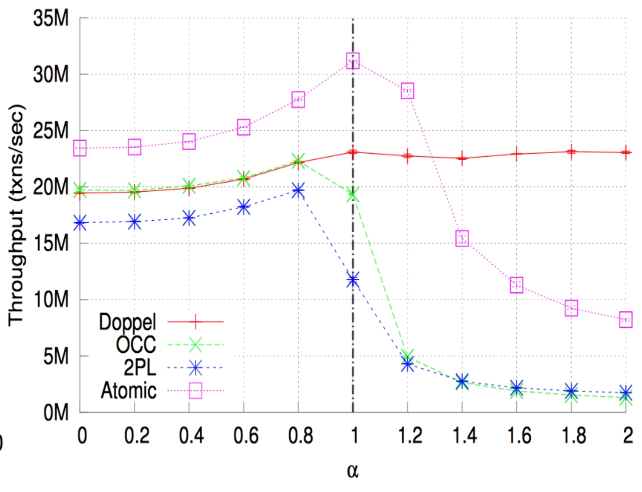
		Linux		Radix tree
	RSS	VMA tree	Page table	(rel. to Linux)
Firefox	352 MB	117 KB	1.5 MB	3.9 MB (2.4×)
Chrome	152 MB	124 KB	1.1 MB	2.4 MB (2.0×)
Apache	16 MB	44 KB	368 KB	616 KB (1.5×)
MySQL	84 MB	18 KB	348 KB	980 KB (2.7×)

Figure 7. Memory usage for alternate VM representations.

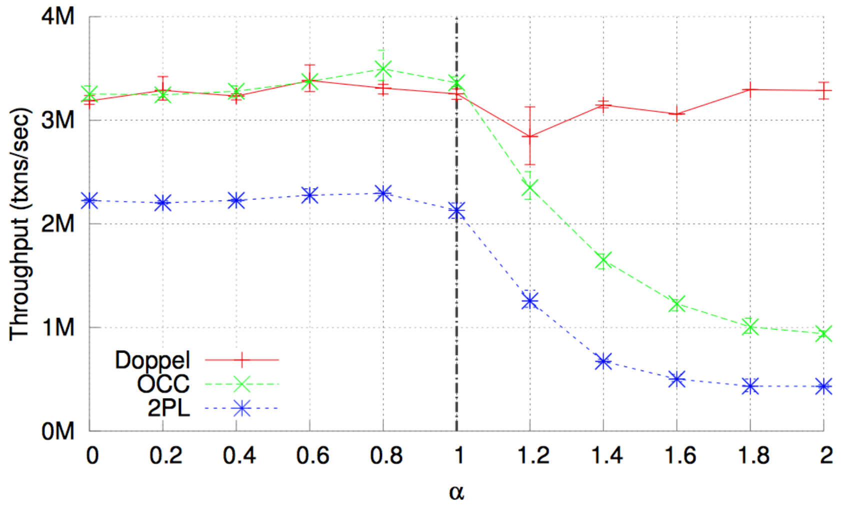
Now, phase reconciliation came in 2014 to take the best of each of these together in phase reconciliation on transactional memory. They were interested in making transactional memory faster and better than 2PL and OCC. They were inspired by the different approaches before them. From Sync-phases, it took the idea of splitting transactions up into computation and commit phases but modified it. Phase reconciliation doesn't split transactions but assign transactions to different phases, based on the type of data they access and the type of operations they perform. From DORA, it took the idea of partitioning data and running one partition on one core but modified it. Phase reconciliation doesn't partition data but partitions local copies of data amongst cores for write and provide a way to re-merge the data for access by other cores. From Doppel, it took the idea of using per core counters and the reconciling the per-core data structure when they execute but makes the performance burden by amortizing the effect of reconciliation over many transactions. The contribution of phase reconciliation was to make these ideas work in a larger transaction system. Figure 8 shows the different comparisons of Doppel, OCC and 2PL. Figure 8 (a) shows the total throughput for INCR1 as a function of the percentage of transactions that increment the single hot key, (b) shows the total throughput for INCRZ as a function of the zipfian distribution parameter ($\propto$)and (c) shows the RUBiS-C benchmark.



(a)



(b)



(c)

Figure 9. Performance test on Doppel, OCC and 2PL

From the Results of the performance test, it is obvious Doppel performed better. In the INCR1 microbenchmark, there are 1M 16-byte keys, and each transaction increments the value of a single key. There is a single popular key and the percentage of transactions, which increment that key is varied; each other transaction randomly chooses from the not- popular keys.

In the INCRZ microbenchmark, there are 1M 16-byte keys. Each transaction increments the value of one key, chosen with ∝ (Zipfian distribution of popularity). The vertical line indicates when Doppel starts splitting keys. Doppel works well in larger transactional system

Conclusion

These papers had one thing in common. They had the same goal, which is to solve the problem of memory contention by processes. Due to the exponential growth of cores on a chip, this problem became more and more critical and had to be addressed. There all tried addressing this problem in different ways each with its stronghold and downside. Phase transaction tried getting the best of the earlier approaches to make transactional memory more efficient. It would have been interesting if there were a comparison of current thread-to-transaction systems and thread-to-data systems to see which one is really better now.

REFERENCES

[1] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-Memory Transactions. In OSDI, pages 511–524. USENIX Association, 2014.

[2] Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. PVLDB , 3(1-2):928–939, 2010.

[3] J. Schneider, F. Landau, and R. Wattenhofer. Synchronization phases (to speed up transactional memory). Technical report, July 2011.

[4] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM:Scalable address spaces for multithreaded applications. InEu-rosys, pages 211–224. ACM, 2013.