Remote Transaction Commit: Centralizing Software Transactional Memory Commits

Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran

Abstract—
Software Transactional Memory (STM) has recently emerged as a promising synchronization abstraction for multicore architectures. State-of-the-art STM algorithms, however, suffer from performance challenges due to contention and spinning on locks during the transaction commit phase. In this paper, we introduce Remote Transaction Commit (or RTC), a mechanism for executing commit phases of STM transactions. RTC dedicates server cores to execute transactional commit phases on behalf of application threads. This approach has two major benefits. First, it decreases the overheads of spinning on locks during commit, such as the number of cache misses, blocking of lock holders, and CAS operations. Second, it enables exploiting the benefits of coarse-grained locking algorithms (simple and fast lock acquisition, reduced false conflicts) and bloom filter-based algorithms (concurrent execution of independent transactions). Our experimental study on a 64-core machine with four sockets shows that RTC solves the problem of performance degradation due to spin locking on both micro-benchmarks (red-black trees), and macro-benchmarks (STAMP), especially when the commit phase is relatively long and when thread contention increases.

Index Terms—Software Transactional Memory, Remote Commit, Transactions dependency

1 Introduction

Software transactional memory [1] (STM) is an appealing concurrent programming abstraction that shifts the burden of synchronization from the programmer to an underlying software framework. With STM, programmers organize reads and writes to shared memory in "atomic blocks", which are guaranteed to satisfy atomicity, consistency, and isolation properties.

Inspired by database transactions, STM manages an atomic block by storing its memory accesses into private structures called read-sets and write-sets. To achieve consistency, a validation mechanism is used (either at encounter time or at commit time) to detect conflicting transactions (i.e., read-write or write-write conflicts). If two transactions conflict, one of them is aborted, and all its prior changes are undone. When a transaction commits, it permanently publishes its writes to the shared memory. That way, other transactions (or at least the successful ones) cannot see its intermediate states, which guarantees atomicity and isolation.

Transactional memory is increasingly gaining traction: Intel has released a C++ compiler with STM support [2]; IBM [3] and Intel [4] have released commodity hardware processors integrating transactional memory features; and GCC has released programming language extensions to support STM [5].

Despite those promising advancements, STM has performance and scalability limitations. They are mainly caused by the overheads of managing meta-data, such as validating read-set entries, locking write-set entries, and rolling-back aborted transactions. Among those overheads, the ones involved in the execution of the transaction commit part have a significant influence on the application performance. In Table 1 of Section 3 we quantify those overheads using the STAMP benchmark [6], and we show that speeding up the commit-time part can significantly enhance the overall performance.

One of the important design decisions that affects the commit-time overhead is the granularity of the locks acquired during commit. Commit-time locking can be coarse-grained, as TML [7] and NOrec [8], compacted using bloom filters [9], as RingSTM [10], or fine-grained using ownership records, as TL2 [11]. In addition, some algorithms replace the commit-time locking with an eager approach where locks are acquired at encounter time (e.g., TinySTM [12]). In general, fine-grained locking, either eager or lazy, decreases the probability of unnecessary serialization of non-conflicting executions with additional costs due to the presence of many meta-data to be handled, and a more complex implementation.

Despite the wide spectrum of STM algorithms provided in literature, almost all of them assume a naive spin-locking (sometimes with back-off delays). In our analysis (in Section 3), we show that the overhead of such a spinning technique forms a significant part of the overall commit-time latency. This overhead becomes dominant in high core count architectures where the slow inter-socket communications contribute with an additional degradation [13].

Motivated by these observations, we present Remote Transaction Commit (or RTC) a mechanism for processing commit parts of STM transactions "remotely". RTC's basic idea is to execute the commit operations of a transaction in dedicated servicing threads. In RTC,

• *Authors are with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24060*

when a client transaction[1] reaches the commit phase, it sends a commit request to a server (potentially more than one). Instead of competing on spinlocks, the servicing threads communicate with client threads through a cache-aligned request array. This approach reduces cache misses (which are often due to the spinning operation on locks), and reduces the number of CAS operations during commit[2]. Additionally, dedicating CPU cores for servers reduces the probability of interleaving the execution of different tasks on those cores due to the OS scheduling. Blocking the execution of commit phase, for allowing other transactions to interleave their processing on the same core, is potentially disruptive for achieving high performance.

In the past, works similar to RTC have been proposed, but none of them has been specifically tailored (and optimized) for STM. In particular, RCL [15] is a locking mechanism based on the idea of executing lock-based critical sections through remote threads. Applying the same idea in STM is appealing because it meets the current technological trend of having many cores available.

In conclusion, RTC can be seen as a deployment of earlier attempts to overcome the overhead of spin locking in lock-based applications by adopting those new techniques inside STM frameworks. RTC follows a direction similar to the one of the lazy, lightweight, coarse-grained STM algorithms, like NOrec [8]. By doing so, it minimizes the number of locks that will be replaced by remote execution (precisely, it replaces only one lock, which is acquired at NOrec's commit). Moreover, RTC solves the problem of coarse-grained algorithms, which serialize independent (i.e., non-conflicting) commit phases due to the single lock, by using additional secondary servers to execute independent commit requests (which do not conflict with the transaction currently committed on the main server).

Our implementation and experimental evaluation show that RTC is particularly effective when transactions are long and the contention is high. If the write-sets of transactions are long, transactions that are executing their commit phases will be a bottleneck. All other spinning transactions will suffer from blocking, cache misses, and unnecessary *CASing*, which are significantly minimized in RTC. In addition, our experiments show that when the number of threads exceeds the number of cores, RTC performs and scales significantly better. This is because, RTC solves the problem of blocking lock holders by an adverse OS schedule that causes chained blocking.

RTC is designed for systems deployed on high core count multicore architectures where reserving few cores for executing those portions of transactions does not significantly impact the overall system's concurrency level. However, the cost of dedicating cores on some architecture may be too high given the limited parallelism. For those architectures, we extend RTC by allowing application threads, rather than a dedicated server thread, to combine the execution of the commit phases. This idea is similar to the *flat combining* [16] approach proposed for concurrent data structures (in fact, we name the new version of RTC as RTC-FC). Our experiments include the comparison between RTC and RTC-FC, thus clarifying the workloads that can benefit more from one approach or the other.

Summarizing, the paper makes the following contributions:

- We introduce RTC, an STM algorithm that uses remote execution of *internal* critical sections (i.e., commit phases) to minimize concurrency overhead, by reducing cache misses, CAS operations, and thread blocking. Furthermore, RTC uses bloom filters to allow the concurrent execution of independent transactions.
- We analyze STAMP applications [6] to show the relationship between commit time ratio and RTC's performance gain.
- Through experimentation, we show that both RTC and RTC-FC have low overhead, peak performance for long running transactions, and significantly improved performance for high number of threads (up to 4x better than other algorithms in high thread count configurations). We also show the impact of increasing the number of secondary servers.

The rest of the paper is organized as follows. In Section 2, we overview past and related work. Section 3 describes RTC's design, Section 4 details the RTC algorithm, and Section 5 analyzes RTC's correctness. Section 6 discusses RTC-FC. We evaluate both RTC and RTC-FC in Section 7, and discuss some possible extensions in Sections 8 and 9. We conclude the paper with Section 10.

2 Past and Related Work

The current STM algorithms cover an array of different decision alternatives on contention management, memory validation, and metadata organization. Also, different STM algorithms are best suited for different workloads. Thus, an interesting research direction has been to combine the benefits of different STM algorithms in an adaptive STM system, which switches at run-time between the algorithms according to the workload at-hand [17].

The approach of dedicating threads for specific tasks has been covered before in both TM [18] and non-TM proposals [19]. RTC is different from them in the role assigned to the dedicated threads.

Orthogonal to STM, hardware transactional memory (HTM) is currently gaining traction as an alternative for efficiently managing transactions on the hardware level [3], [4]. However, all the released HTM architectures are best-effort (i.e. there is no guarantee of an

1. We will call an RTC servicing thread as "server" and an application thread as "client".

2. It is well understood that spinning on locks and increased usage of CAS operations can seriously hamper application performance [14], especially in multicore architecture.

eventual commit), which means that there is still a need to have an efficient software fall-back path. Orthogonal to transactional memory as a whole, other directions in literature target solving the overhead of accessing the shared memory by adapting the architectures [20], proposing new programming models [21], involving the compiler [22]. Despite the scalability limitations of STM compared to these alternatives, it remains as a promising alternative which provides simple, composable, and programmable, model for designing (reasonably efficient) large and complex transactional applications on the current architectures.

In the following subsections, we overview RCL and past STM algorithms that are relevant for RTC, and contrast them with RTC itself.

2.1 Remote Core Locking

Remote Core Locking (RCL) [15] is a mutual exclusion locking mechanism used for synchronizing cooperating threads. The main idea of RCL is to dedicate some cores to execute critical sections. If a thread reaches a critical section, it will send a request to a server thread using a cache-aligned requests array. An earlier similar idea is Flat Combining [16], which dynamically elects one client to temporarily take the role of the server, instead of dedicating servicing threads.

Unlike STM, both the number of locks and the logic of the critical sections vary according to the applications. Thus, RCL client's request must include more information than RTC, like the address of the lock associated with the critical section, the address of the function that encapsulates the client's critical section, and the variables referenced or updated inside the critical section. Re-engineering, which in this case means replacing critical sections with remote procedure calls, is also required and made off-line using a refactoring mechanism [23].

RCL outperforms traditional locking algorithms like MCS [24] and Flat Combining [16] in legacy applications with long critical sections. This improvement is due to three main enhancements: reducing cache misses on spin locks, reducing time-consuming CAS operations, and ensuring that servers that are executing critical sections are not blocked by the scheduler.

On the other hand, RCL has some limitations. Handling generic lock-based applications, with the possibility of nested locks and conditional locking, puts extra obligations on servers. RCL must ensure livelock freedom in these cases, which complicates its mechanism and requires thread management. Also, legacy applications must be re-engineered so that critical sections can be executed as remote procedures. This specific problem cannot exist in STM because the main goal of STM is to make concurrency control transparent from programmers. Finally, it is not clear how sequence locks (i.e., locks with version numbers), which are used by most STM algorithms, can be replaced with RCL. As we will show later, RTC does not suffer from these limitations, while still retaining all the benefits of RCL.

2.2 STM Algorithms

NOrec [8] is a lazy STM algorithm which uses minimal metadata. Only one global timestamped lock is acquired at commit time to avoid write-after-write hazards. Validation on NOrec is value-based, which reduces false conflicts. Each transaction validates its read-set after each read and extends its local timestamp if validation succeeds. At commit time, write-set is published on shared memory. RTC inherits all the strong properties of NOrec (e.g., privatization safety, reduced false conflicts, and minimal locking overhead), and also solves the problem of executing independent commit phases serially, by adding dependency detector servers. Moreover, RTC inherits the easy integration with hardware transactions, as we show in more details in Section 9.

RingSTM [10] introduced the idea of detecting conflicts using bloom filters [9]. Each thread locally keeps two bloom filters, which represent the thread's read-set and write-set. All writing transactions first join a shared ring data structure with their local bloom filters. Readers validate a new read against the bloom filters of writing transactions, which join the ring after the transaction starts. The main difference between RingSTM and RTC is in the way they use bloom filters. RingSTM uses bloom filters to validate read-sets and synchronize writers, which increases false conflicts according to the size of the bloom filter. In RTC, as we show later, bloom filters are only used to detect dependency between transactions, while validation is still done at the memory level.

TL2 [11] is also an appealing STM algorithm which uses ownership records. However, this fine-grained speculation is not compatible with the RTC idea of remote execution. In addition, the native version of TL2 does not have the same strong properties of NOrec, such as privatization safety. Finally, despite the fine-grained locking approach used in TL2, all writing transactions eventually modify a shared global timestamp. This issue is partially addressed in SkyTM [25], a further extension of TL2 that employs scalable visible readers instead of relying on a single global timestamp.

3 Remote Transaction Commit

3.1 Design

The basic idea of RTC is to execute the commit phase of a transaction in a dedicated main server core, and to detect non-conflicting pending transactions in another secondary server core. This way, if a processor contains n cores, two cores will be dedicated as servers, and the remaining $n - 2$ cores will be assigned to clients. For this reason, RTC is more effective when the number of cores is large enough to afford dedicating two of them as servers. However, the core count in modern architectures is increasing, so that reserving two cores does not represent a limitation for RTC applicability.

As stated in Section 1, the architecture of RTC can be considered as an extension of NOrec. Figure 1 shows the structure of a NOrec transaction. A transaction can be

seen as the composition of three main parts: initialization, body, and commit. The initialization part adjusts the local variables at the beginning of the transaction. In the transaction body, a set of speculative reads and writes are executed. During each read, the local read-set is validated to detect conflicting writes of concurrent transactions, and, if the validation is successful, the new read is added to the read-set. Writes are also saved in local write-sets to be published at commit. During the commit phase, the read-set is repeatedly validated until the transaction acquires the lock (by an atomic CAS operation to increase the global timestamp). The write-set is then published into the shared memory, and finally, the global lock is released.

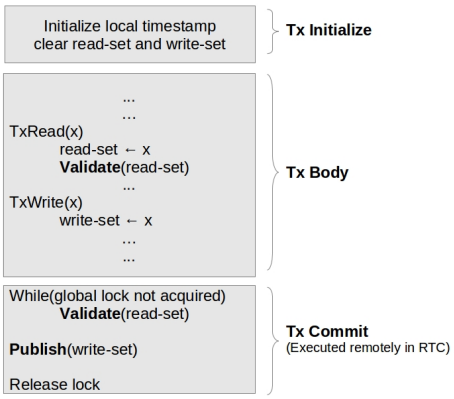

Fig. 1. Structure of a NOrec transaction.

This well defined mechanism of NOrec can be converted to remotely executing the transaction commit part. Unlike lock-based applications, which contain programmer-defined locks with generic critical sections, RTC knows precisely the number of locks to acquire (i.e., only one global lock), when to execute the commit (i.e., at the transaction end), and what to execute inside the commit (i.e., validating transaction read-set and publishing its write-set). This simplifies the role of the servers, in contrast to server-based optimizations for locks such as RCL [15] and Flat Combining [16], which need additional mechanisms (either by re-engineering as in RCL or at run-time as in Flat Combining) to indicate the procedures to execute in behalf of the clients to the servers.

RTC is therefore simple: clients communicate with servers (either main or secondary) using a cache-aligned requests array to reduce caching overhead. A client's commit request always contains a reference to the transactional context (read-set and write-set, local timestamp, and bloom filters). This context is attached to the transaction request when it begins. A client starts its commit request by changing a $state$ field in the request to a pending state, and then spins on this $state$ field until the server finishes the execution of its commit and resets the $state$ field again. On the server side, the main server loops on the array of commit requests until it finds a client with a pending state. The server then obtains the transaction's context and executes the commit. While the main server is executing a request, the secondary server also loops on the same array, searching for independent requests. Note that it does not execute any client requests unless the main server is executing another non-conflicting request.

Figure 2 illustrates the flow of execution in both NOrec and RTC. Assume we have three transactions. Transaction T_A is a long running transaction with a large write-set. Transaction T_B does not conflict with T_A and can be executed concurrently, while transaction T_C is conflicting with T_A. Figure 2(a) shows how NOrec executes these three transactions. If T_A acquires the lock first, then both T_B and T_C will spin on the shared lock until T_A completes its work and releases the lock, even if they can run concurrently. Spinning on the same lock results in significant number of useless CAS operations and cache misses. Moreover, if T_A is blocked by the OS scheduler, then both the spinning transactions will also wait until T_A resumes, paying an additional cost. This possibility of OS blocking increases with the number of busy-waiting transactions.

In Figure 2(b), RTC moves the execution of T_A's commit to the main server. Transactions T_B and T_C send a commit request to the server and then spin on their own requests (instead of spinning on a shared global lock) until they receive a reply. During T_A's execution, the secondary server (which is dedicated to detecting dependency) discovers that T_B can run concurrently with T_A, so it starts executing T_B without waiting for T_A to finish. Moreover, when T_A is blocked by the OS scheduler, this blocking does not affect the execution of its commit on the main server, and does not block other transactions. Blocking of the servers is much less frequent here, because client transactions are not allowed to execute on server cores.

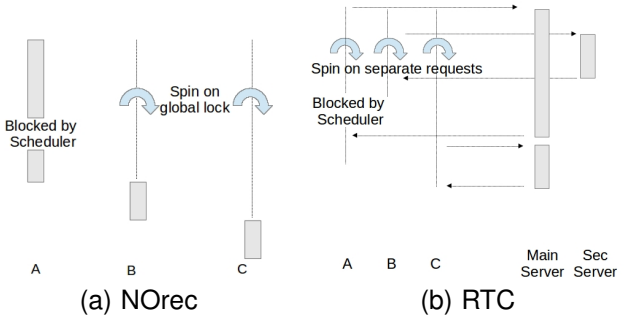

Fig. 2. Flow of commit execution in NOrec and RTC.

3.2 Dependency Detection

RTC leverages a secondary server to solve the problem of unnecessary serialization of independent commit requests. The secondary server uses bloom filters [9] to detect dependency between transactions. Each transaction

Benchmark	8 threads		16 threads		32 threads		48 threads	
	%trans	%total	%trans	%total	%trans	%total	%trans	%total
genome	49	32	53	14	54	5	56	3
intruder	25	19	37	31	39	26	19	9
kmeans	43	34	56	27	60	15	62	11
labyrinth	0	0	0	0	0	0	0	0
ssca2	83	53	94	63	95	66	92	39
vacation	6	5	17	16	42	36	50	45

TABLE 1
Ratio of NOrec's commit time in STAMP benchmarks.

keeps two local bloom filters and updates them at each read/write (in addition to updating the normal read-set and write-set). The first one is a write filter, which represents the transaction writes, and the second one is a read-write filter, which represents the union of the transaction reads and writes. If the read-write filter of a transaction T_X does not intersect with the write filter of a transaction T_Y currently executed in the main server, then it is safe to execute T_X in the secondary server. (We provide a proof of the independence between T_X and T_Y in Section 5).

Synchronization between threads in NOrec is done using a single global sequence lock. Although RTC needs more complex synchronization between the main server and the secondary server, in addition to synchronization with the clients, we provide a lightweight synchronization mechanism that is basically based on the same global sequence lock, and one extra servers lock. This way, we retain the same simplicity of NOrec's synchronization. (Section 4 details RTC's synchronization).

The effectiveness of the secondary server is evident when the write-set size is large. The secondary server adds synchronization overhead to the main server. This overhead will be relatively small if commit phase is long, and transactions are mostly independent. On the other hand, if write-sets are short (indicating short commit phases), then even if transactions are independent, the time taken by the secondary server to detect such independent transactions is long enough so that the main server may finish its execution before the secondary server makes a substantial progress. To solve this issue, RTC dynamically enables/disables the secondary server according to the size of the transaction write-set. The secondary server works on detecting non-conflicting transactions when the write-set size exceeds a certain threshold (In Section 8, we show an experimental analysis of this threshold). As a consequence, the interactions between main and secondary servers are minimized so that the performance of the transactions executed in the main server (that represents the critical path in RTC) is not affected.

Another trade off for RTC is the bloom filter size. If it is too small, many false conflicts will occur and the detector will not be effective. On the other hand, large bloom filters need large time to be accessed. Bloom filter access must be fast enough to be fruitful. In our experiments, we used the same size as in the RSTM default configuration (1024 bits), as we found that other sizes give similar or worse performance.

3.3 Analysis of NOrec Commit Time

As we mentioned before, RTC is more effective if the commit phase is not too short. Table 1 provides an analysis of NOrec's commit time ratio of the STAMP benchmarks [6][3]. In this experiment, we measure the commit time as the sum of the time taken to acquire the lock and the time taken for executing the commit procedure itself. We calculated both *A)* the ratio of commit time to the transactions execution time ($\%trans$), and *B)* the ratio of commit time to the total application time ($\%total$). The results show that the commit time is already predominant in most of the STAMP benchmarks. The percentage of commit time increases when the number of threads increases (even if the $\%total$ decreases, which means that the non-transactional execution increases). This means that the overhead of acquiring the lock and executing commit becomes more significant in the transactional parts.

As our experimental results show in Section 7, the increase of RTC performance is proportional to the commit latency. In fact, benchmarks with higher percentage of commit time (*genome*, *ssca2*, and *kmeans*) gain more from RTC than the others with small commit execution time (*intruder* and *vacation*) because the latter do not take advantages from the secondary server. However, they still gain from the main server, especially when the number of transactions increases and competition between transactions on the same lock becomes a significant overhead. Benchmarks with a dominating non-transactional workloads (*labyrinth*) show no difference between NOrec and RTC because no operations are done for those non-transactional parts during the commit phase.

4 RTC Algorithm

The main routines of RTC are servers loops and the new commit procedure. The initialization procedure and transaction body code are straightforward, thus we only

3. We excluded *yada* here and in all our further experiments as it evidenced errors (segmentation fault) when we tested them on RSTM, even in the already existing algorithms like NOrec. We also excluded *bayes* because its performance varies significantly between runs, so it is not useful for benchmarking.

briefly discuss them. Our full implementation, examples and test-scripts used for experimental evaluation, are available for reviewers as a supplemental material.

4.1 RTC Clients

Client commit requests are triggered using a cache-aligned requests array. Each commit request contains three items:

- **state.** This item has three values. READY means that the client is not currently executing commit. PENDING indicates a commit request that is not handled by a server yet. ABORTED is used by the server to inform the client that the transaction must abort.
- **Tx.** This is the reference to the client's transactional context. Basically, servers need the following information from the context of a transaction: *read-set* to be validated, *write-set* to be published, the *local timestamp*, which is used during validation, and *filters*, which are used by the secondary server.
- **pad.** This is used to align the request to the cache line (doing so decreases false sharing).

RTC initialization has two main obligations. The first one is allocating the requests array and starting the servers. The second one is to set the affinity of the servers to their reserved cores.

When a transaction begins, it is bound to the clients' $cpuset$ to prevent execution on server cores (note that it's allowed to bound more than one client to the same core, according to the scheduler). It also inserts the reference of its context in the requests array. Finally, the local timestamp is assigned to the recent consistent global timestamp. Reads and writes update the bloom filters in addition to their trivial updates of read-sets and write-sets. Reads update the read-write filter, and writes update both the write filter and the read-write filter.

Client post validation is value-based like NOrec. Algorithm 1 shows how it generally works. In lines 3-4, transaction takes a snapshot of the global timestamp and loops until it becomes even (meaning that there is no commit phase currently running on both main and secondary servers). Then, read-set entries are validated (line 5). Finally, the global timestamp is read again to make sure that nothing is modified by another transaction while the transaction was validating (Lines 6-9).

Servers need also to validate the read-set before publishing the write-set. The main difference between server validation and client validation is that there is no need to check the timestamp by the server, because the main server is the only thread that changes the timestamp.

Algorithm 2 shows the commit procedure of RTC clients. Read-only transactions do not need to acquire any locks and their commit phase is straightforward. A read-write transaction starts its commit phase by validating its read-set to reduce the overhead on servers if it is already invalid (line 5). If validation succeeds, it changes its state to PENDING (line 7). Then it loops until one of the servers handles its commit request and changes the state to either READY or ABORTED (line 8). It will either commit or roll-back according to the reply (lines 9–12). Finally, the transaction clears its bloom filters for reusing them (line 13).

Algorithm 1 RTC: client validation

```
1:  procedure CLIENT-VALIDATION
2:      t = global_timestamp
3:      if t is odd then
4:          retry validation
5:      Validate read-set values
6:      if t ≠ global_timestamp then
7:          retry validation
8:      else
9:          return t
10: end procedure
```

Algorithm 2 RTC: client commit

```
1:  procedure COMMIT
2:      if read_only then
3:          ...
4:      else
5:          if ¬ Client-Validate(Tx) then
6:              TxAbort()
7:          req.state = PENDING
8:          loop while req.state ∉ (READY, ABORTED)
9:          if req.state = ABORTED then
10:             TxAbort()
11:         else
12:             TxCommit()
13:         ResetFilters()
14: end procedure
```

4.2 Main Server

The main server is responsible for executing the commit part of any pending transaction. Algorithm 3 shows the main server loop. By default, the dependency detection (DD) is disabled. The main server keeps looping on client requests until it reaches a PENDING request (line 6). Then it validates the client read-set. If validation fails, the server changes the state to ABORTED and continues searching for another request. If validation succeeds, it starts the commit operation in either DD-enabled or DD-disabled mode according to a threshold of the client write-set size (lines 7–14).

Execution of the commit phase without enabling DD is straightforward. The timestamp is increased (which becomes odd, indicating that the servers are working), the write-set is published to memory, the timestamp is then increased again (to be even), and finally the request state is modified to be READY.

When DD is enabled, synchronization between the servers is handled using a shared $servers\_lock$. First, the main server informs the secondary server about its current request number (line 23). Then, the global timestamp is increased (line 24). The order of these two lines is important to ensure synchronization between the main and secondary servers. The main server must also acquire $servers\_lock$ before it increments the timestamp again at the end of the commit phase (lines 26–29) to prevent the main server from proceeding until the

Algorithm 3 Main server loop. Server commit with dependency detection disabled/enabled.

```
1:  procedure MAIN SERVER LOOP
2:      DD = false
3:      while true do
4:          for i ← 1, num_transactions do
5:              req ← req_array[i]
6:              if req.state = PENDING then
7:                  if ¬ Server-Validate(req.Tx) then
8:                      req.state = ABORTED
9:                  else if write_set_size < t then
10:                     Commit(DD-Disabled)
11:                 else
12:                     DD = true
13:                     Commit(DD-Enabled)
14:                     DD = false
15: end procedure
16: procedure COMMIT(DD-Disabled, req)
17:     global_timestamp++
18:     WriteInMemory(req.Tx.writes)
19:     global_timestamp++
20:     req.state = READY
21: end procedure

22: procedure COMMIT(DD-Enabled, req, i)
23:     mainreq = req_array[i]
24:     global_timestamp++
25:     WriteInMemory(req.Tx.writes)
26:     loop while !CAS(servers_lock, false, true)
27:     global_timestamp++
28:     mainreq = NULL
29:     servers_lock = false
30:     req.state = READY
31: end procedure
```

secondary server finishes its work. As we will show in the correctness part (in Section 5), this $servers\_lock$ guarantees that the secondary server will only execute as an extension of the main server's execution. Comparing the two algorithms, we see that DD adds only one CAS operation on a variable, which is (only) shared between the servers. Also, DD is not enabled unless the write-set size exceeds the threshold. Thus, the overhead of DD is minimal.

4.3 Secondary Server

Algorithm 4 shows the secondary server's operation. It behaves similar to the main server except that it does not handle PENDING requests unless it detects that:

- DD is enabled (line 4);
- Timestamp is odd, which means that main server is executing a commit request (line 6);
- The new request is independent from the current request handled by the main server (line 9).

The commit procedure is shown in lines 12–26. Validation is done before acquiring $servers\_lock$ to reduce the time of holding the lock (lines 13–14). However, since it is running concurrently with the main server, the secondary server has to validate that the main server is still handling the same request (line 16) after acquiring $servers\_lock$. This means that, even if the secondary server reads any false information from the above three points, it will detect that by observing either a different timestamp or a NULL $mainreq$ after the acquisition of $servers\_lock$. The next step is to either commit or abort according to its earlier validation (lines 18-24). Finally,

Algorithm 4 RTC: secondary server

```
1:  procedure SECONDARY SERVER LOOP
2:      while true do
3:          for i ← 1, num_transactions do
4:              if DD = false then continue
5:              s = global_timestamp
6:              if s&1 = 0 then continue
7:              req ← req_array[i]
8:              if req.state = PENDING then
9:                  if Independent(req,mainreq) then
10:                     Commit(Secondary)
11: end procedure

12: procedure COMMIT SECONDARY(req)
13:     if ¬ Server-Validate(req.Tx) then
14:         aborted = true
15:     if CAS(servers_lock, false, true) then
16:         if s <> global_timestamp or mainreq = NULL then
17:             servers_lock = false
18:         else if aborted = true then
19:             req.state = ABORTED
20:             servers_lock = false
21:         else
22:             WriteInMemory(req.Tx.writes)
23:             req.state = READY
24:             servers_lock = false
25:             loop while global_timestamp = s
26: end procedure
```

in case of commit, secondary server loops until the main server finishes its execution and increases the timestamp (line 25). This is important to prevent handling another request, which may be independent from the main server's request but not independent from the earlier request.

The secondary server does not need to change the global timestamp. Only the main server increases it at the beginning and at the end of its execution. All pending clients will not make any progress until the main server changes the timestamp to an even number, and the main server will not do so until the secondary server finishes its work (because if the secondary server is executing a commit phase, it will be holding $servers\_lock$).

5 CORRECTNESS

To prove the correctness of RTC, we first show that there are no race conditions impacting RTC's correctness when the secondary server is disabled. Then, we prove that our approach of using bloom filters guarantees that transactions executed on the main and secondary servers are independent. Finally, we show how adding a secondary server does not affect race-freedom between the main and the secondary server, or between clients and servers [4].

RTC with DD Disabled: With the secondary server disabled, RTC correctness is similar to that of NOrec. Briefly, the post validation in Algorithm 1 ensures that: *i)* no client is validating while server is committing, and *ii)* each transaction sees a consistent state after each read.

4. In all of the proof arguments, we assume that instructions are executed in the same order as shown in Section 4's algorithms – i.e., sequential consistency is assumed. We ensure this in our C/C++ implementation by using memory fence instructions when necessary (to prevent out-of-order execution), and by using volatile variables when necessary (to prevent compiler re-ordering).

The only difference between NOrec and RTC without dependency detection is in the way they increment the timestamp. Unlike NOrec, there is no need to use the CAS operation to increase the global timestamp, because no thread is increasing it except the main server. All commit phases are executed serially on the main server, which guarantees no write conflicts during commit, either on the timestamp or on the memory locations themselves.

Transaction Independence: The secondary server adds the possibility of executing two independent transactions concurrently. To achieve that, each transaction keeps two bloom filters locally: a write-filter "$wf(t)$", which is a bitwise representation of the transaction write-set, and a read-write filter "$rwf(t)$", which represents the union of its read-set and write-set. Concurrent commit routines (in both main and secondary servers) are guaranteed to be independent using these bloom filters. We can state that: if a transaction T_1 is running on the RTC main server, and there is a pending transaction T_2 such that $rwf(T_2) \cap wf(T_1) = \emptyset$, then T_2 is independent from T_1 and can run concurrently using the secondary server. This can be proven as follows: T_1 does not increase the timestamp unless it finishes validation of its read-set. Thus, T_2 will not start unless T_1 is guaranteed to commit. Since $rwf(T_2) \cap wf(T_1) = \emptyset$, T_1 can be serialized before T_2. T_1 cannot invalidate T_2 because T_1's write-set does not intersect with T_2's read-set. The write-after-write hazard also cannot happen because the write filters are not intersecting. If T_2 aborts because of an invalidated read-set, it will not affect T_1's execution.

RTC with DD Enabled: Finally, we prove that transaction execution is still race-free when the secondary server is enabled. Synchronization between the main and the secondary server is guaranteed using the $servers\_lock$. The main server acquires the lock before finishing the transaction (clearing $mainreq$ and incrementing the global timestamp) to ensure that the secondary server is idle. The secondary server acquires the $servers\_lock$ before starting, and then it validates both $mainreq$ and the timestamp. If they are invalid, the secondary server will not continue, and will release the $servers\_lock$, because it means that the main server finishes its work and starts to search for another request.

More detailed, in lines 26-29 of Algorithm 3, the main server increments the timestamp in a mutually exclusive way with lines 15-24 of the secondary server execution in Algorithm 4 (because both parts are enclosed by acquiring and releasing the $servers\_lock$). Following all possible race conditions between line 27 of Algorithm 3 and the execution of the secondary server shows that the servers' executions are race-free. Specifically, there are four possible cases for the secondary server when the main server reaches line 27 (incrementing the timestamp after finishing execution):

- Case 1: before the secondary server takes a snapshot of the global timestamp (before line 5). In this case, the secondary server will detect that the main server is no longer executing any commit phase. This is because, the secondary server will read the new (even) timestamp, and will not continue because of the validation in line 6.
- Case 2: after the secondary server takes the snapshot and before it acquires the $servers\_lock$ (after line 5 and before line 15). In this case, whatever the secondary server will detect during this period, once it tries to acquire the $servers\_lock$, it will wait for the main server to release it. This means that, after the secondary server acquires the $servers\_lock$, it will detect that the timestamp is changed (line 16) and it will not continue.
- Case 3: after the secondary server acquires the $servers\_lock$ and before this lock is released (after line 15 and before line 24). This cannot happen because the $servers\_lock$ guarantees that these two parts are mutually exclusive. So, in this case, the main server will keep looping until the secondary server finishes execution and releases the $servers\_lock$.
- Case 4: after the secondary server releases the $servers\_lock$ (after line 24). This scenario is the only scenario in which the main and secondary servers are executing commit phases concurrently. Figure 3 shows this scenario. In this case, the secondary server works only as an extension of the currently executed transaction in the main server, and the main server cannot finish execution and increment the timestamp unless the secondary server also finishes execution.

Thus, the main server will not continue searching for another request until the secondary server finishes its execution of any independent request. Line 25 of Algorithm 4 guarantees the same behavior in the other direction. If the secondary server finishes execution first, it will keep looping until the main server also finishes its execution and increments the timestamp (which means that the secondary server executes only one independent commit request per each commit execution on the main server).

In conclusion, the RTC servers provide the same semantics of single lock STM algorithms. Although two independent commit blocks can be concurrently executed on the main and secondary servers, they appear to other transactions as if they are only one transaction because they are synchronized using different locks (not the global timestamp). Figure 3 shows that the global timestamp increment (to be odd and then even) encapsulates the execution of both the main and the secondary servers. This also guarantees that the clients' validations will not have race-conditions with the secondary server, because clients are not validating in the period when the timestamp is odd.

Using a single lock means that RTC is privatization-safe, because writes are atomically published at commit time. The value-based validation minimizes false conflicts and enables the detection of non-transactional writes. Finally, servers repeatedly iterate on clients requests, which guarantees livelock-freedom and more fair progress of transactions.

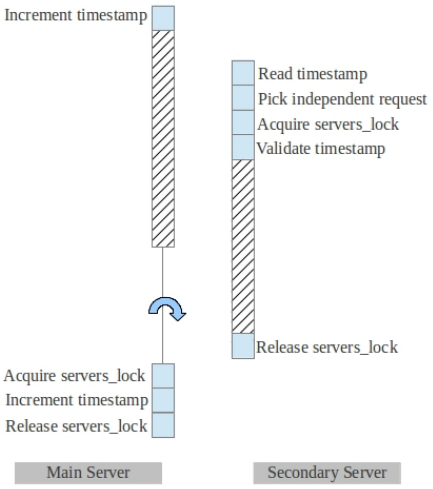

Fig. 3. Flow of commit execution in the main and secondary servers. Even if the main server finishes execution before the secondary server, it will wait until the secondary server releases the $servers\_lock$.

6 RTC WITH FLAT-COMBINING

The approach of dedicating cores for executing server threads has its own cost because it disallows cores from running application threads and it enforces data accessed by transactions to be cached on those cores. As we show later in Section 7, in most workloads this cost is dominated by the benefits of reducing cache misses, reducing CAS operations, and preventing lock holders from being descheduled. However, in some architectures, especially those with low core count, this cost may become notable and impact the performance. To address this issue, we extend RTC by introducing RTC-FC, a version of RTC with no dedicated cores for servers.

The only distinguishing point between RTC-FC and RTC is on the assignment of the thread that plays the role of the server. In RTC-FC, we use an idea similar to *flat combining* [16] that selects one of the running clients to combine the requests of pending clients. To do that, each client changes its request status from `READY` to `PENDING`, and then it tries to be the combiner (using one CAS operation on a `combiners-lock`). If it succeeds, it executes one server iteration, serving all the `PENDING` requests, including its own request, similar to an iteration of Algorithm 3 with *DD-DISABLED*. If the CAS fails, this means that another thread became the combiner, thus, the thread spins on its request status similarly to RTC. During its spinning, it periodically checks if the current combiner releases the `combiners-lock`. If so, it retries to be the combiner.

Although RTC-FC avoids dedicating cores for servers, it has the following overheads: first, it adds at least one more CAS operation for each transaction; second, it increases the probability of descheduling the `combiners-lock` holder; and finally, it obligates threads to pause their executions while servicing other requests, which also adds an overhead due to caching the data of other requests. The problem of descheduling a combiner can be partially solved by enforcing the clients that fail to CAS the `combiners-lock` to call $sched\_yield$ in order to give up their OS time slice instead of spinning. The effect of overloading the combiner cache with the data of other transactions can also be alleviated by using a NUMA-aware flat-combining algorithm similar to [26], where the combiner executes the commit phases of transactions belonging only to its NUMA-zone in order to exploit the locality of this NUMA-zone. In the next section we show how those parameters affect the performance of RTC-FC, and discuss the cases in which RTC-FC fits best.

7 EXPERIMENTAL EVALUATION

We implemented RTC in C++ (compiled with gcc 4.6) and integrated into the RSTM framework [27] (compiled using default configurations). Our experiments are performed on a 64-core AMD Opteron machine (128GB RAM, 2.2 GHz, 64K of L1 cache, 2M of L2 cache) with 4 sockets and 16 cores per socket (with 2 NUMA-zones per physical socket, making a total of 8 NUMA-zones). Each NUMA-zone has a 6M of L3 cache.

Our benchmarks for evaluation included micro-benchmarks (such as red-black tree and linked list), and the STAMP benchmark suite [6]. We also evaluated multi-programming cases, where the number of transactions is more than the number of cores. Our competitors include NOrec (as representative of approaches relying on global metadata), RingSW (because it uses bloom filter), and TL2 (representing an ownership-record based approach). We used a privatization-safe version of TL2 for a fair comparison. All the STM algorithms and the benchmarks used are those available in RSTM. All reported data points are averages of 5 runs.

7.1 Red-Black Tree

Figure 4 shows the throughput of RTC, RTC-FC, and their competitors on a red-black tree with 1M elements and a delay of 100 no-ops between transactions. In Figure 4(a), when 50% of operations are reads, all algorithms scale similarly (RTC-FC is slightly better than the others), but both versions of RTC sustain high throughput, while other algorithms' throughput degrades. This is a direct result of the cache-aligned communication among transactions. In high thread count, RTC is slightly better than RTC-FC because of the overheads of the latter as discussed in Section 6. In Figure 4(b), when 80% of the operations are reads, the degradation point of all algorithms shifts (to the right) because contention is lower. However, RTC scales better and reaches peak performance when contention increases. At high thread count, RTC improves over the best competitor by 60% in the first case and 20% in the second one.

Additionally, we focused on making a connection between the performance (in terms of throughput) and the

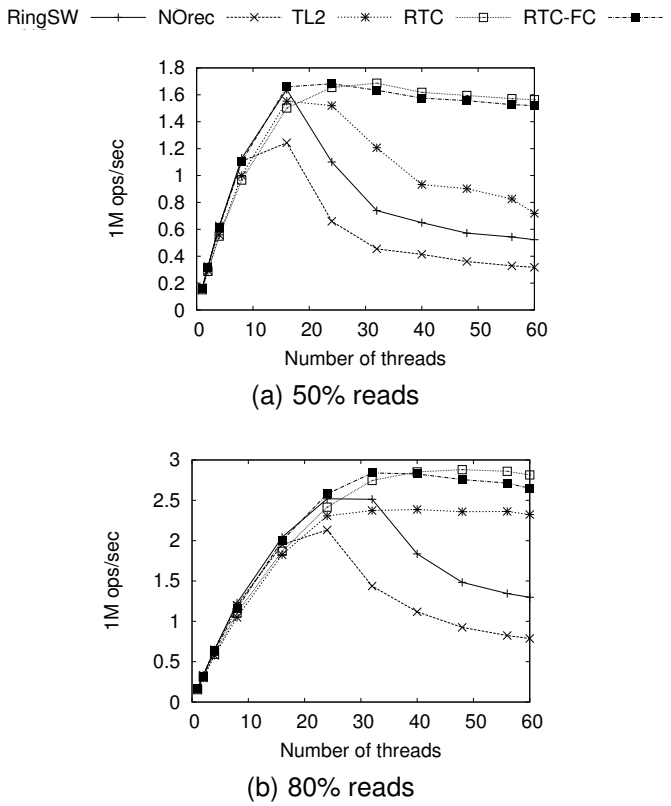

Fig. 4. Throughput (per micro-second) on red-black tree with 1M elements.

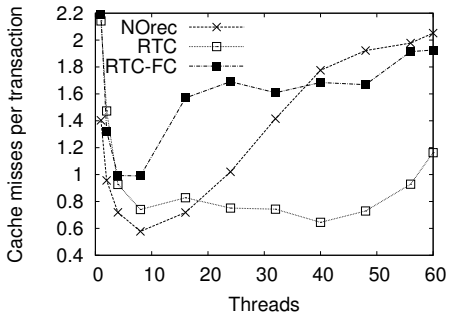

Fig. 5. Cache misses per transaction on red-black tree with 1M elements.

average number of cache misses per transaction generate by NOrec and RTC. Figure 5 shows the results. At high number of threads, the number of cache misses per transaction on NOrec is higher than RTC. Comparing Figures 4(a) and 5, an interesting connection can be found between the point in which the performance of NOrec starts to drop and the point in which the number of NOrec's cache misses starts to increase. This comparison clearly points out the impact of RTC design on decreasing cache misses due to spin locks. RTC-FC on the other hand, suffers from more cache-misses. However, those misses are not generated because of spinning. Rather, they are generated because each client is playing the role of the server frequently, and thus it is obligated to validate the read-set of the other clients, which may result in evicting its own data from its cache. Summarizing, although the number of cache misses is not the only parameter that affects the performance of RTC, Figure 5 gives an important reasoning about the effect of RTC and RTC-FC on the cache misses on both the actual data and the meta-data. Such a comparison allows for a better understanding of their behavior.

In the next experiment, we created up to 256 threads[5] and repeated the experiment while progressively enabling the cores of only one socket (16 cores), two sockets (32 cores), and the whole four sockets (64 cores). Our goal in this experiment is to make a stress test to reveal the side effect of having server-client communication in heavy loads, as well as to seek RTC's performance saturation point. More specifically, we use the approach of enabling/disabling CPU sockets to analyze the effect of *i)* running very large number of concurrent threads on few number of cores, while dedicating two of them as servers, and *ii)* having inter-socket synchronization rather than intra-socket synchronization.

Figure 6 shows the results in a red-black tree with 50% reads. In Figure 6(a), when only one socket is enabled, all the transactions execute on one socket, which decreases the overhead of cache misses and CAS operations. For this reason, both versions of RTC cannot gain a lot from the efficient remote core locking mechanism, and thus the gap between them and the other algorithms is small. Additionally, dedicating two cores out of sixteen as servers in RTC has a significant effect on the overall performance. That is why in this case, RTC-FC performs better than RTC. In Figure 6(b), when the number of cores becomes 32 (on 2 sockets), the penalty of dedicating two cores for RTC decreases, thus RTC and RTC-FC perform similarly. At the same time, the overheads in the other algorithms increase because meta-data now are cached in two sockets rather than one. As a result, the overall performance of RTC/RTC-FC increases compared to the other STM algorithms. The performance improvement continues in the last case (Figure 6(c)), when the number of cores becomes 64 (on 4 sockets). Specifically, starting from 32 threads, RTC/RTC-FC perform better than the best competitor by an average of 3x at high thread count. Our analysis confirms previous studies, which conclude that cross-socket sharing should be minimized as it is one of the performance killers [13]. Also, in this case, RTC becomes better than RTC-FC because the overhead of dedicating cores is minimized while the overhead of overloading the clients with the combiner tasks increases.

Figure 6 also shows that in the multi-programming case (when threads become more than cores), RTC's performance starts to slightly degrade like the other STM algorithms. However, this degradation is the normal degradation due to the contention on the shared red-black tree, which confirms that RTC solves the issue of

5. This is the maximum number of threads allowed by RSTM.

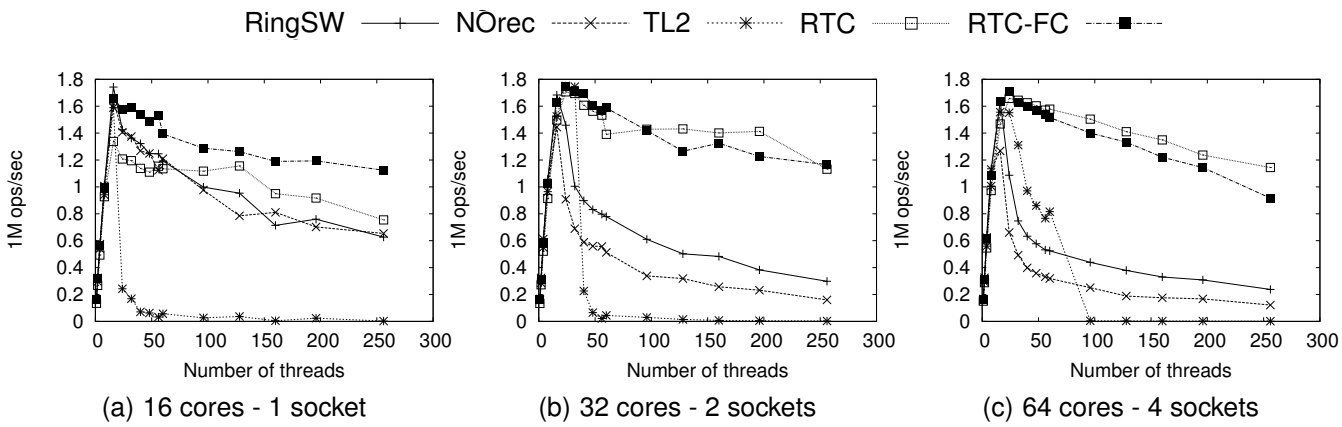

Fig. 6. Throughput on red-black tree with 1M elements, 100 no-ops between transactions, 50% reads.

spin locking and leaves the overhead of STM framework limited to the contention on the application-level data.

To conclude, RTC still has some limitations, like the effect of dedicating cores for servers, and the normal contention on the application-level data (which is not targeted by RTC's mechanism). However, when the number of cores increases, these negative effects are dominated by the improvements due to the optimized locking mechanism.

7.2 Linked List

In Figure 7(a), we show the results using the linked list benchmark. It represents the worst case workload for RTC/RTC-FC and we include it in order to show the RTC design's limitations in unfavorable scenarios. Linked list is a benchmark which exposes non optimal characteristics in terms of validation and commit. In fact, in a doubly linked list with only 500 nodes, each transaction makes on average hundreds of reads to traverse the list, and then it executes few writes (two writes in our case) to add or remove the new node. This means that the read-set size is too large compared to the write-set size. Since RTC servers have to re-validate the read-set of the clients before publishing the write-set, pulling a large read-set like that affects the performance significantly and nullifies any gains from optimizing the actual commit phase (which mainly consists of acquiring the global timestamp and publishing the write-set). Figure 7(b) confirms that by showing the cache-misses per transaction in that case. Here, the cache misses saved by the cache-aligned communication are clearly dominated by thrashing the cache of the servers with the read-sets of the clients. The problem increases in RTC-FC since the combiners are actual client threads that may sacrifice their own cached data to validate the reads-set of the other clients. It is worth to note that this issue does not occur for read-only workloads, because RTC does not involve servers in executing read-only transactions.

As a solution to this issue, an STM runtime can be made to heuristically detect these cases of RTC degradation by comparing the sizes of read-sets and write-sets, and switching at run-time from/to another appropriate algorithm as needed. Earlier work proposes a lightweight adaptive STM framework [17]. In this framework, the switch between algorithms is done in a "stop-the world" manner, in which new transactions are blocked from starting until the current in-flight transactions commit (or abort) and then switch takes place. RTC can be easily integrated in such a framework. Switching to RTC only requires allocating the requests array and binding the servers and clients to their $cpusets$ (which can be achieved using C/C++ APIs). Switching away from RTC requires terminating the server threads and deallocating the requests array.

7.3 STAMP

Figure 8 shows the results for six STAMP benchmarks, which represent more realistic workloads with different attributes. It is important to relate these results to the commit time analysis in Table 1. RTC has more impact when commit time is relatively large, especially when the number of threads increases.

In four out of these six benchmarks (*ssca2*, *kmeans*, *genome*, and *labyrinth*), RTC has the best performance when the number of threads exceeds 20. Moreover, for *kmeans* and *ssca2*, which have the largest commit overhead according to Table 1, RTC has better performance than all algorithms even at low number of threads. For *labyrinth*, RTC performs the same as NOrec and TL2, and better than RingSW. This is because, RTC does not have any overhead on non-transactional parts, which dominate in *labyrinth*. Also, in all cases, RTC outperforms NOrec at high number of threads. Even for *vacation* and *intruder*, where the commit time percentage is small (6%–50% and 19%–39%, respectively), RTC outperforms NOrec and has the same performance as RingSW for high number of cores. In those two benchmarks, TL2 is the best algorithm (especially for low thread count) because they represent low-contention workloads, where serializing the (mostly independent) commit phases, as in NOrec and RTC, affects the performance. For RTC

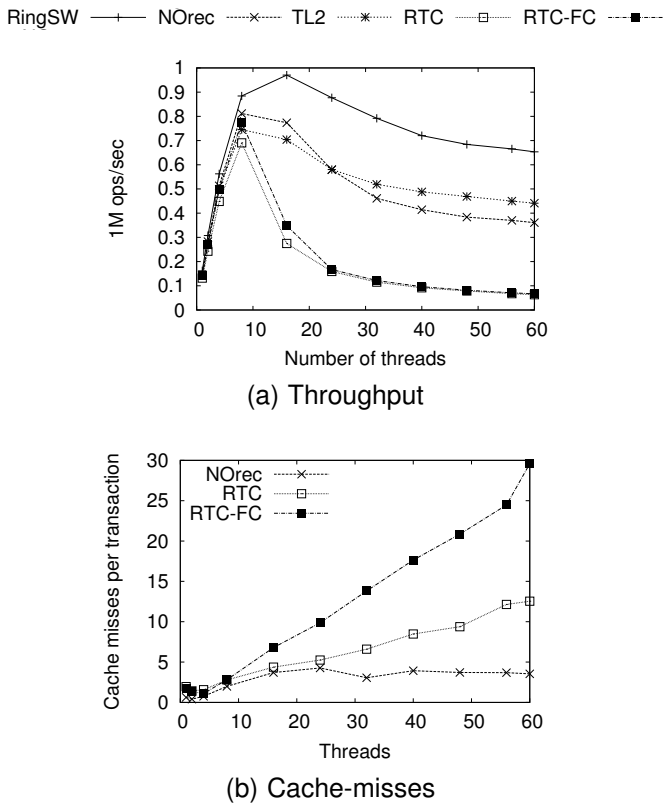

Fig. 7. Throughput and cache-misses per transaction on doubly linked list with 500 elements, 50% reads, and 100 no-ops between transactions.

specifically, like linked list, scalability in those benchmarks is clearly affected by the small ratio of the commit phase (as mentioned in Table 1).

In general, both RTC and RTC-FC perform similarly. However, RTC-FC performs better in the low-contention workloads (e.g. *genome* and *vacation*), especially for small thread count (less than 8). This gap decreases at high number of threads (more than 8), because the benefit of dedicating cores for servers in RTC (e.g., disallowing lock holder descheduling) increases.

8 Extending RTC with more servers

The current implementation of RTC uses only two servers: one main server and one secondary server. It is easy to show that using one main server is reasonable. This is because we replace only one global lock (in NOrec) with a remote execution. Even if we add more main servers, their executions will be serialized because of this global locking. Adding more secondary servers (which search for independent requests) is, however, reasonable. This is because it may increase the probability of finding such an independent request in a reasonable time, which increases the benefits from secondary servers. However, leveraging on a fine grain performance analysis, we decided to tune RTC with only one secondary server. This decision is supported by the results obtained by running RTC with more secondary servers, which are shown in Figure 9. They highlight that the synchronization overhead needed for managing the concurrent execution of more secondary servers, is higher than the gain achieved.

In Figure 9, N reads and N writes of a large array's elements are executed in a transaction, which results in write-sets of size N. The writes are either totally dependent by enforcing at least one write to be shared among transactions, or independent by making totally random reads and writes in a very long array. Figure 9 shows that the overhead of adding another secondary server is more than its gain. Performance enhancement using one DD is either the same or even better than using two DD in all cases. The same conclusion holds for executing more than one commit phase on the secondary server in parallel with the same main server's commit. Although both enhancements should have a positive effect in some theoretical cases of very long main server commits, we believe that in practical cases, like what we analyzed, the gain is limited.

We also used this experiment to determine the best threshold of the write-set size after which we should enable dependency detection. The time taken by the main server to finish the commit phase is proportional to the size of the write-set (read-set size is not a parameter because validation is made before increasing the timestamp). Thus, small write-sets will not allow the secondary server to work efficiently and will likely add unnecessary overhead (putting into consideration that the time taken by the secondary server to detect independent transactions does not depend on the transaction size because bloom filters are of constant size and they are scanned in almost constant time). To solve this problem, RTC activates the secondary server only when the size of the write-set exceeds a certain threshold.

In case of dependent transactions, the dependency detector (DD) cannot enhance performance because it will not detect a single independent transaction. Note that, the overhead of DD does not exceed 5% though, and it also decreases when the write-set size increases (reaches 0.5% when the size is 50). When transactions are independent, DD starts to yield significant performance improvement when the write-set size reaches 20 elements (obtains 30% improvement when size is 40). Before 10 elements, DD's overhead is larger than the gain from concurrent execution, which results in an overall performance degradation.

We also calculated the number of transactions which are executed on the secondary server. In all the independent cases, it varies from 2% to 11%. Since the percentage of improvement is higher in most cases, this means that DD also saves extra time by selecting the most appropriate transaction to execute among the pending transactions, which reduces the probability of abort.

According to these results, we use only one secondary server, and we select a threshold of 20 elements to enable the secondary server in our experiments.

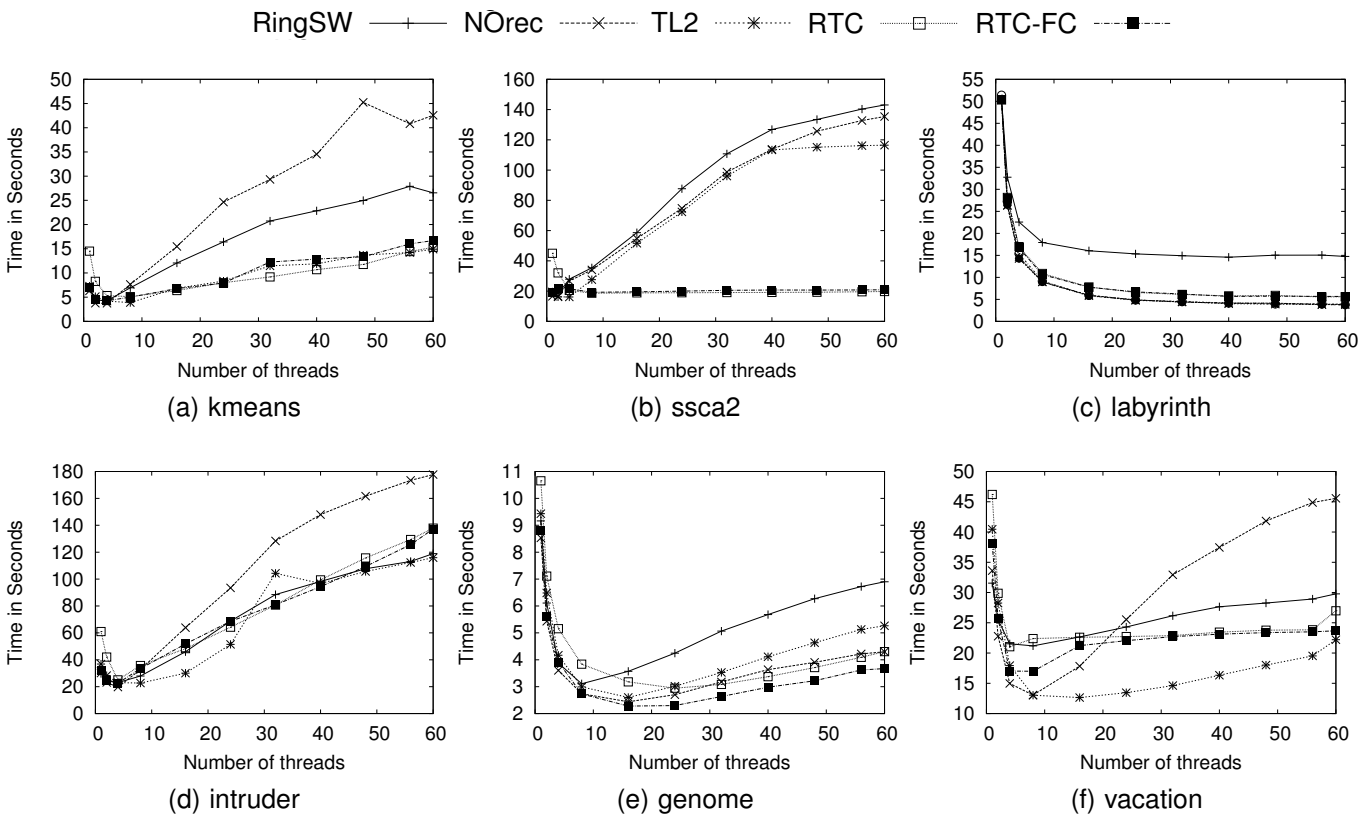

Fig. 8. Execution time on STAMP benchmark suite's applications.

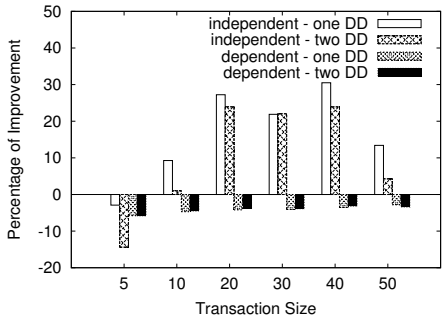

Fig. 9. Effect of adding dependency detector servers.

9 Using RTC in Hybrid TM

NOrec has been successfully used as a fallback path to *best-effort* HTM transactions [28], [29], [30]. This is because it uses only one global timestamp as a shared meta-data. Replacing NOrec with RTC in such hybrid algorithms is a feasible extension to our work. To do so, no modification is needed at the client (software) execution because HTM transactions only need to know whether there is a software transaction executing its commit phase or not (which would be done by the servers exploiting the global timestamp). Moreover, centralizing the commit phases in the servers allows for more optimizations on the hybrid algorithms themselves, such as exploiting servers for profiling the HTM execution.

10 Conclusions

Software transactional memory is a highly promising synchronization abstraction, but state-of-the-art STM algorithms are plagued by performance and scalability challenges. Analysis of these STM algorithms on the STAMP benchmark suite shows that transaction commit phases are one of the main sources of STM overhead. RTC reduces this overhead with a simple idea: execute the commit phase in a dedicated servicing thread. This reduces cache misses, spinning on locks, CAS operations, and thread blocking. Our implementation and evaluation shows that the idea is very effective – up to 4x improvement over state-of-the-art STM algorithms in high thread count.

RTC builds upon similar ideas on remote/server thread execution previously studied in the literature, most notably, Flat Combining and RCL. However, one cannot simply apply them to an STM framework as is. In one sense, our work shows that, this line of reasoning is effective for improving STM performance.

Acknowledgment

Authors would like to thank the invaluable comments of anonymous IEEE TC reviewers. This work is supported in part by US National Science Foundation under grant CNS-1217385, and by US Air Force Office of Scientific Research under grant FA9550-14-1-0187.

REFERENCES

[1] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," Synthesis Lectures on Computer Architecture, vol. 5, no. 1, pp. 1–263, 2010.

[2] Intel Corporation, "Intel C++ STM Compiler 4.0, Prototype Edition," http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/, 2009.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in ISCA, 2013, pp. 225–236.

[4] J. Reinders, "Transactional synchronization in Haswell," http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/, 2013.

[5] TM Specication Drafting Group, "Draft specification of transactional language constructs for c++, version 1.1," 2012.

[6] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. IEEE, 2008, pp. 35–46.

[7] M. Spear, A. Shriraman, L. Dalessandro, and M. Scott, "Transactional mutex locks," in SIGPLAN Workshop on Transactional Computing, 2009.

[8] L. Dalessandro, M. Spear, and M. Scott, "Norec: streamlining stm by abolishing ownership records," in ACM Sigplan Notices, vol. 45, no. 5. ACM, 2010, pp. 67–78.

[9] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.

[10] M. Spear, M. Michael, and C. von Praun, "Ringstm: scalable transactions with a single atomic instruction," in Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. ACM, 2008, pp. 275–284.

[11] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," Distributed Computing, pp. 194–208, 2006.

[12] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures. ACM, 2007, pp. 221–228.

[13] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013, pp. 33–48.

[14] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012.

[15] J. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications," in Work in progress in the Symposium on Operating Systems Principles, SOSP, vol. 11, 2011.

[16] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures. ACM, 2010, pp. 355–364.

[17] M. Spear, "Lightweight, robust adaptivity for software transactional memory," in Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures. ACM, 2010, pp. 273–283.

[18] G. Kestor, R. Gioiosa, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero, "Stm2: A parallel stm for high performance simultaneous multithreading systems," in International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011. IEEE, 2011, pp. 221–231.

[19] C. B. Zilles, J. S. Emer, and G. S. Sohi, "The use of multithreading for exception handling," in Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, 1999, pp. 219–229.

[20] M. Stonebraker, "The case for shared nothing," IEEE Database Eng. Bull., vol. 9, no. 1, pp. 4–9, 1986.

[21] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands et al., "Productivity and performance using partitioned global address space languages," in Proceedings of the 2007 international workshop on Parallel symbolic computation. ACM, 2007, pp. 24–32.

[22] M. Bond, M. Kulkarni, M. Salmi, M. Zhang, S. Biswas, J. Huang, and A. Sengupta, "Octet: Practical concurrency control for dynamic analyses and systems."

[23] M. Fowler and K. Beck, Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[24] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Transactions on Computer Systems (TOCS), vol. 9, no. 1, pp. 21–65, 1991.

[25] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in TRANSACT, 2009.

[26] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012, 2012, pp. 257–266.

[27] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), 2006.

[28] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory," in Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS XVI, 2011, pp. 39–52.

[29] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: the importance of nonspeculative operations," in SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011), 2011, pp. 53–64.

[30] A. Matveev and N. Shavit, "Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory," in Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '15. ACM, 2015, pp. 59–71.

Ahmed Hassan received the BSc in computer science and the MSc in computer engineering at Alexandria University, Egypt. He is currently a PhD student at Virginia Tech. His research interests include transactional memory, concurrent data structures, and distributed computing.

Roberto Palmieri received the BSc in computer engineering, MSc and PhD degree in computer science at Sapienza, University of Rome, Italy. He is a Research Assistant Professor in the ECE Department at Virginia Tech. His research interests include exploring concurrency control protocols for multicore systems, cluster and geographically distributed systems, with high programmability, scalability, and dependability.

Binoy Ravindran is a Professor of Electrical and Computer Engineering at Virginia Tech, where he leads the Systems Software Research Group, which conducts research on operating systems, run-times, middleware, compilers, distributed systems, fault-tolerance, concurrency, and real-time systems. Ravindran and his students have published more than 220 papers in these spaces, and some of his group's results have been transitioned to the DOD. Dr. Ravindran is an Office of Naval Research Faculty Fellow and an ACM Distinguished Scientist.