NEMO: NUMA-aware Concurrency Control for Scalable Transactional Memory

Mohamed Mohamedin
Virginia Tech
Blacksburg, Virginia
mohamedin@vt.edu

Sebastiano Peluso
Virginia Tech
Blacksburg, Virginia
peluso@vt.edu

Masoomeh Javidi Kishi
Lehigh University
Bethlehem, Pennsylvania
maj717@lehigh.edu

Ahmed Hassan
Alexandria University
Alexandria, Egypt
ahmed.hassan@alexu.edu.eg

Roberto Palmieri
Lehigh University
Bethlehem, Pennsylvania
palmieri@lehigh.edu

ABSTRACT

In this paper we present NEMO, a NUMA-aware Transactional Memory (TM) design and implementation optimized for promoting scalability in applications running on top of NUMA architectures. NEMO deploys a hybrid design where conflicting threads alternate the usage of single timestamps and vector clocks to identify inconsistent executions depending upon the source of conflict. We assessed the performance of NEMO by using both synthetic and well-known OLTP transactional workloads. Our approach offers improvements over the six state-of-the-art competitors we implemented.

ACM Reference Format:
Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri. 2018. NEMO: NUMA-aware Concurrency Control for Scalable Transactional Memory. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225123

1 INTRODUCTION

Transactional Memory (TM) [15] is a powerful programming abstraction for implementing parallel and concurrent applications. TM frees programmers from the complexity of managing multiple threads that access the same set of shared objects. The advent of multicore architectures, which provides (sometimes massive) parallel computing capabilities for thread execution, clearly favored the diffusion of TM. Today, this hardware is widely available on the open market; even non-expensive processors are equipped with tens of physical core, improving real parallelism.

The growing number of cores per processor led designers to produce architectures where the whole address space is divided into multiple slices (or zones). In these architectures, the latency to perform a memory access varies depending on the processor in which the thread executes, and the physical segment of memory that stores the requested memory location. Such a design, called Non-Uniform Memory Access (NUMA) [21]), is the de-facto standard for multicore platforms that possess high parallel computing capability.

There is a class of workloads where application data can be partitioned to provide scalability when running on NUMA architectures. In this class of applications, data can be organized such that an object is placed in a zone close to the thread that accesses the object. This organization fits the NUMA design where *(i)* a processor is physically connected to one memory zone, which offers very low access latency to that zone, and (ii) where the latency to access other zones is higher. We name this class of workloads as *scalable workloads*. It should be legit for programmers to expect the performance of these applications to scale up when they increase the number of threads physically executing in parallel.

In this paper, we study the performance of such *scalable workloads* when executed using TM frameworks on NUMA architectures. Our main observation is that, although many STM algorithms with different designs have been proposed since the adoption of TM, most of them are not designed to scale in NUMA because of the inherent synchronization they use for updating metadata. In fact, metadata handling in those algorithms is not optimized for NUMA architectures, which hampers. scalability.

To quantitatively support the above claim, we conduct an evaluation study consisting of the following two major tests (a detailed description and plots is reported in Section 2).

We first assess the performance of current TM algorithms in NUMA architectures. To do so, we deploy several state-of-the-art TM algorithms over an AMD 64-core machine equipped with 4 physical sockets, each of which hosts a 16-core processor. The memory is physically partitioned into 8 NUMA zones; each of these directly interfaces with 8 cores. The lesson learned from this test is that letting non-conflicting threads access shared metadata causes traffic on the physical bus interconnecting different processor sockets (and thus NUMA zones). Synchronizing metadata is a common technique to allow transactions to efficiently identify inconsistent operations. Unfortunately, given the constraints of NUMA architectures, updating metadata becomes the bottleneck when the workload is mostly non-conflicting (or scalable as defined above).

Addressing this issue requires modifying the way metadata are accessed in TM algorithms. Our second test aims at understanding the performance gains that can be achieved by doing so. Specifically, in this test we identify the inherent cost of using a NUMA

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225123

architecture when shared variables are used to manage the synchronization of parallel threads. The results of this test clarify that when threads deployed on the same socket cooperate using shared variables stored in the NUMA zone connected to that socket, the bottleneck of handling shared metadata no longer exists due to the absence of traffic over the inter-sockets bus.

We use the above observations as the design principles of a new TM algorithm, which we name Nemo[1]. Nemo is scalable - in the presence of a scalable workload: its performance increases when the number of threads deployed on different cores increases. The core idea of Nemo is to treat conflicts involving transactions that execute within the same socket differently from those that involve transactions on different sockets (or NUMA zones[2]).

Nemo has been implemented in C++ and evaluated using benchmarks (e.g., TPC-C [31]) configured to provide scalable workloads. Our findings show that Nemo is the only solution that continues to offer increasing performance beyond the threshold corresponding to the number of cores enclosed in a single socket. Specifically, Nemo outperforms TLC [2] and TL2-GV5 [10], which were designed to alleviate the pressure of updates on global shared metadata, and all other STM approaches that we tested.

2 MOTIVATIONS FOR A NUMA-AWARE STM

The Non-Uniform Memory Access (NUMA) design is the de-facto standard for interfacing hardware threads with the main memory in highly parallel multicore architectures. In a NUMA design, one memory socket (i.e., a memory chip that constitutes a part of the overall system memory) is physically attached with one processor socket (or, if the socket is capable of maintaining multi-dies, then one die inside the socket), and it represents the so called NUMA zone. We say that a thread executing on a particular socket accesses a *local* NUMA zone when it performs a memory operation on a location that is maintained within the NUMA zone connected to that socket. Otherwise, we say that the thread accesses a *remote* NUMA zone.

When a hardware thread accesses a memory location whose address is located in the local NUMA zone, the latency is very small (e.g., 9 *nsec* using DDR3-2000 memory) and the access is performed without contention on the shared bus resource. On the other hand, if the memory location is stored in a remote NUMA zone, the hardware thread is forced to use the shared bus that interconnects all of the sockets in order to fetch the desired value. The latter access is clearly slower than the former. Importantly, if two threads operating in two different NUMA zones work on data stored in their own local NUMA zones, they can proceed in parallel without contending on any shared hardware component.

A typical hardware architecture of the widely used AMD 64-core commercially-available server is structured in 4 sockets of 2 dies each, with 8 cores per die. A NUMA zone is mapped to a die, and therefore the overall number of zones is equal to 8.

On this machine, we performed the tests briefly described in Section 1. In the first test we deployed a version of the Bank benchmark (see Section 7 for additional details about the benchmark) where all of the *accounts* are partitioned across NUMA zones and application threads operate only on accounts stored in their local NUMA zone. This workload matches our definition of scalable. As TM algorithms, we implemented and adopted TL2 [10], SwissTM [11], TinySTM [13], RingSTM [30], and NOrec [7]. TL2, SwissTM, and TinySTM use different conflict detection policies, but all of them lock the objects to be updated individually, by relying on a shared lock table partitioned across NUMA zones. Conversely, NOrec protects the transaction commit phase using a single shared lock, while RingSTM uses a global shared ring data structure to catch and abort invalid executions.

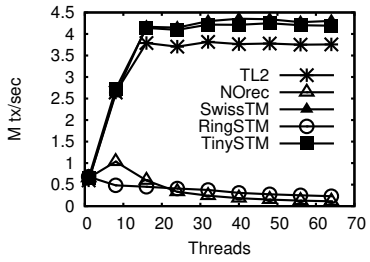

Figure 1: Bank benchmark configured for producing a scalable workload (disjoint transactional accesses).

The results of this experiment are in Figure 1. Here we measured the throughput by varying the number of worker threads. The algorithms using per-object locks provide better performance than the others because they allow for more concurrency in the system, which definitely pays off under a scalable workload. Beside the individual performance of each competitor, we can notice that all the algorithms stop scaling after 16 threads, namely they are not able to provide higher throughput as the number of threads increases beyond that configuration. That configuration represents the maximum number of parallel threads allowed within a single socket. After that point, the cost of updating global metadata becomes very high as this operation likely involves traversing the shared bus that connects different sockets, thus hampering the overall scalability. As an example of that, SwissTM relies on a single timestamp to validate the transactions' read operations. This timestamp is incremented when a write transaction commits, and this represents a high contention bottleneck.

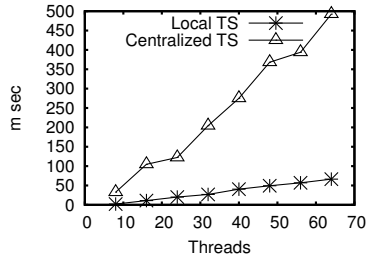

Figure 2: Average cost of 100k increment operations of a centralized vs local timestamp.

[1]A poster version of this paper is available at [23]

[2]In the rest of the paper we use the terms "socket" and "NUMA zone" interchangeably.

The second experiment shows the latency needed for incrementing a logically shared timestamp via a *compare-and-swap* (CAS) primitive. Two configurations are deployed: one uses a single timestamp located in one NUMA zone and all application threads increment that; the other one uses 8 physical timestamps, which are distributed across the 8 zones, and the application threads running in a zone are allowed to increment only the timestamp that is located in their local zone. The results in Figure 2 show the average time (in milliseconds) required to perform 100k increments by varying the number of threads per NUMA zone. For fairness, we enforced that all zones are loaded equally, meaning the data point at 8 threads represents the configuration with 1 thread per NUMA zone executing increment operations, the one at 16 threads represents the configuration with 2 threads per NUMA zone, and so on up to 64 threads.

The results reveal that updating a single timestamp provides almost no scalability due to the high traffic generated on the shared bus among sockets. On the other hand, the cost of updating a timestamp located in a local zone is very low, even if we adopt the CAS primitive, and therefore the configuration with multiple physical timestamps provides better scalability with a slight increase of latency when increasing the number of threads. It is worth noticing that, in the runs adopting the latter configuration, at any point in time there are always 8 timestamps getting updated in parallel using atomic operations. This consideration is important because it shows that, even though there is contention locally at each zone, the hardware is able to handle it locally at each socket without significant impact on the work performed on other sockets.

The results detailed in this section form the basis of Nemo, our solution for providing scalable performance in the presence of scalable workloads. To accomplish such a goal, two principles should be taken into account: *(i)* threads executing on data stored in different NUMA zones should not interfere with each other if the transactions they are executing do not manifest any conflict; *(ii)* threads executing on the same socket are allowed to share information to validate their executions faster, as this cooperation will not affect the performance of threads executing on other sockets.

3 RELATED WORK

With the advent of NUMA, some of the most important software to adapt include operating systems, system libraries, middleware, and database management systems. Current database management systems perform badly on multicore machines especially NUMA-based machines [14, 18, 25, 28]. In Multimed [28], authors showed that treating today's multicore machines as distributed systems in the design of a database engine, provides much better performance. In their evaluation study, deploying multiple replicated instances of the database engine on the same machine performs better that deploying only a single instance that uses all available cores. The authors of [25] reached a similar conclusion: shared-nothing deployments perform better than cooperative ones.

In [18], the authors showed that allocating memory based on data partitioning, and grouping worker threads improve the performance. This new configuration exploits the locality features of NUMA architectures.

There are proposals that targeted eliminating the bottleneck of any centralized timestamp that is present in most of TM implementations. In the algorithm presented in [26], the timestamp is replaced by a physical (hardware) clock or a set of synchronized physical clocks, and a similar idea of exploiting hardware clocks has been also explored in the design of the TM presented in [27]. However, algorithms based on physical clocks are not expected to scale well as the hardware itself cannot keep a large number of clocks synchronized without paying a significant overhead.

The authors of SkySTM [19] presented a scalable STM algorithm that is privatization safe [29] as well. SkySTM is based on semi-visible reads, which are implemented by using a Scalable NonZero Indicator (SNZI) [12]. Semi-visible reads allow to notify the existence of concurrent readers without knowing which are those readers, and that is a sufficient requirement to avoid the usage of any global timestamp. The performance provided by SkySTM showed a good scalability of the approach, but anyway it still performs slower than a more scalable version of the well-known TL2 algorithm [10], i.e., TL2-GV6. This is mainly due to the fact that SkySTM is privatization safe while TL2 is not. Nemo as well is not privatization safe as it focuses on achieving the maximum performance possible under scalable workloads.

In [20], a NUMA-aware TM is introduced. The basic idea is to change the conflict detector for inter-NUMA zone conflicts such that it is lazy and latency-based, while to still adopt an eager conflict detection policy for intra-NUMA zone conflicts. The results showed an improved performance but limited scalability. Further, a NUMA-aware design has been also adopted in the TM presented in [22], which shares some design choices with Nemo.

Nemo design is also related to the Lock Cohorting technique [4, 9], which proposes an approach to convert different types of locks into NUMA-aware locks. In the experimental evaluation of Nemo we compared our solution to a modified version of the NORec TM [7] where the shared global lock was implemented by following the Lock Cohorting technique. This showed a boost in performance of the original NORec algorithm, although still absent scalability due to the inherent limitations of the sequential commit phase enforced by the algorithm.

An important property that can enable the achievement of scalability in TM is Disjoint-Access Parallelism (DAP) [17]. The DAP property also works very well given the characteristics of a NUMA architecture, which encourages to limit inter-NUMA zone communications as much as possible in order to achieve high performance. As an example, the strongest version of DAP only allows transactions that access common transactional objects to also conflicts on any other shared memory area. Thus, DAP-compliant TM algorithms can provide good scalability on NUMA architectures [8]. Examples include TLC [2], DSTM [15], PermiSTM [1], and the TM presented in [24].

TLC is the most interesting among them since, unlike DSTM, it implements the strongest version of DAP, and, unlike the remaining of them, it is not a pure theoretical algorithm. The main idea of TLC, which can be applied to other timestamp-based STMs (e.g., TL2 [10], TinySTM [13]), is to remove any global timestamp, and to replace it with thread-local timestamps, plus thread-local caches storing a local view of the other threads' timestamps. A thread-local cache is only updated on demand, whenever an outdated value is detected

upon a conflict. TLC suffers from a large number of unnecessary aborts due to outdated cached copies of timestamps, and can only work well under low levels of contention. Nemo shares some of the TLC principles, but it also overcomes the above TLC's disadvantages by exploiting a less conservative read policy, which adheres to the Virtual World Consistency [16] correctness level.

In TrC-MC [5], the authors extended the TLC algorithm as follows: first, they proposed a NUMA-zone level cache (zone partitioning) similar to Nemo's clocks. Second, they used the timestamp extension mechanism, which revalidates a transaction's read-set upon a read to check that returning the current value of an object does not violate the target correctness guarantee. In the design of Nemo, we evaluated such a timestamp extension mechanism based on the revalidation of the read-set, but it showed a performance degradation although it reduced the number of false aborts.

4 DESIGN OF NEMO

Based on our observations on the current NUMA machines as described in Section 2, a NUMA-aware TM algorithm should avoid any centralized shared metadata and should limit data transfers among NUMA zones. In addition, we performed specific tests to show the interference among atomic primitives (e.g., CAS) executed on different NUMA zones in parallel, and we found that each NUMA zone can locally handle the execution of those primitives without any scalability bottleneck. Thus, our basic idea is to use a traditional centralized-like STM algorithm within the boundaries of a NUMA zone, and restrict inter-NUMA zone communications to when they are actually needed — namely, when a transaction requests to access an object stored in another NUMA zone.

This approach of using two different schemes is similar to the Globally Asynchronous Locally Synchronous (GALS) [6] hardware design principle. GALS relaxes the synchrony assumption by having synchronous islands communicating among each other asynchronously. Similarly, we relax the condition of having a single synchronized global timestamp and use synchronous islands where each of them has its own synchronized timestamp. Communication among these islands is asynchronous such that the correctness of the executions across the islands is anyway preserved.

Nemo applies this idea by implementing an intra-NUMA zone concurrency control similar to TL2 [10], and an inter-NUMA zone one whose skeleton resembles the TLC algorithm [2]. In addition, the same design principles adopted for the implementation of the clocks are also adopted for any metadata the concurrency control relies on: specifically, metadata associated with an object is kept in the same NUMA zone where the object is stored (e.g., partitioning of the lock table according to the objects' distribution). Indeed, paying an expensive cost to access metadata in a remote NUMA zone has to be amortized by the possibility of finding the corresponding requested object in that zone.

Nemo maintains a separate logical clock (or timestamp) for each NUMA zone, called *local clock*. Local transactions, which are those that work only on a single NUMA zone, use only that zone's local clock as main reference to correctly execute read operations and mark any new committed snapshot on that zone (no inter-NUMA zone communication is needed in such a case). To lower the number of unnecessary aborts (e.g., the ones enforced by TLC as soon as a thread detects the existence of a new timestamp produced by another thread) each zone also maintains a local view of the other zones' local clock, which can be possibly outdated. At high level we can see a zone identified by a unique integer i as locally storing a vector clock VCZ_i where: $VCZ_i[i]$ is zone i's local clock; and $VCZ_i[j]$ is the current knowledge of zone i about zone j's local clock for any $0 \leq j \neq i$.

Nemo implements *invisible* read operations, meaning write transactions do not know about the existence of any concurrent and conflicting reader on their written objects[3].

Each object stored in a zone is associated with a logical timestamp, which is the value of that zone's local clock at the time the last transaction writing that object committed. In addition, each transaction keeps a set of all read objects, called its *read-set*. Like in other STMs, a transaction's read-set is used to validate the execution of the transaction right before the transaction commits: this is done by checking that the timestamp associated with an object did not change since the object was read by the transaction.

As we will better understand in Section 6, in Nemo that validation is required for write transactions as well as read-only ones (i.e., transactions that do not execute any write operation), due to the lazy and more permissive scheme adopted to synchronize transactions executing on different zones. Indeed, even though a transaction is guaranteed to observe a correct snapshot at any time of its execution, that observation might not be compatible with any other observation performed by other committed transactions, and therefore an abort might be required to preserve safety for the histories of transactions committed by Nemo. This means that Nemo trades the maximum level of correctness for scalability, i.e., while the histories of committed transactions are still guaranteed to be opaque, any history only preserves the well-known virtual world consistency criterion [16] (which is weaker than opacity).

The write operations performed by a transaction are simply buffered in the transaction's *write-set*, which is used as a *redo log* to apply the changes in memory when the transaction is guaranteed to commit. At commit time, a transaction acquires all locks associated with the objects in the write-set; it validates the read-set as described above; it writes back the whole write-set to the main memory; it updates the vector clocks of the NUMA zones whose memory has been updated to reflect the creation of a new snapshot on those zones; it updates the timestamp of the updated objects, and it releases the previously acquired locks. Failures on either locks acquisition or validation cause a transaction to abort and restart.

The way the per-zone vector clocks are updated during the commit of a write transaction is of fundamentally importance to maintain the scalability of the approach. A first version of Nemo adopts a lock-based mechanism that is proved to be non scalable for configuration with more than 40 threads in our experimental evaluation; then Nemo also provides a lock-free mechanism that is proved to overcome the limitations of the previous one.

Although Nemo does not rely on a single global clock as a reference to correctly serialize the read operations of a transaction at the absolute time the transaction begins, it leverages the per-zone vector clocks to incrementally build the transactions serialization point. This is similar in spirit to the TLC algorithm but, unlike TLC,

[3]We use the term *object* to refer to memory locations.

our scheme is designed to significant reduce the number of unnecessary aborts: intuitively a running transaction keeps track of the newest local timestamp it is allowed to observe for each zone it read from so far; and its read operation for an object o with timestamp t on a new zone j returns o's value only if there is not the possibility that the snapshot with local timestamp t on j overwrote any of the already observed timestamps on other zones. This is done without revalidating the transaction's read-set, and by only looking at j's vector clock, i.e., VCZ_j.

In any case, when a read operation from a zone i to a zone j is issued, if the observed timestamp on j is greater than the local view that zone i has about zone j (i.e., the timestamp is greater than $VCZ_i[j]$), then zone i advances its view accordingly.

Another important issue to address in the design of Nemo was the way the memory is allocated. In order to maintain data locality, newly allocated memory must be placed on a specific NUMA zone (e.g., adding a new element to a linked list in zone i must allocate the new list node on i's memory). Nemo provides a custom memory allocator to properly organize an application's shared memory.

5 DETAILS OF NEMO

In the following we provide the details of the Nemo concurrency control. We first describe the data structures that support the execution of transactions under Nemo, and then we describe the behavior of each transactional operation, by supporting our description with the corresponding pseudocode.

5.1 Data Structures

Nemo distinguishes between the data structures having a *thread-level scope* and the ones having a *system-level scope*. The former category includes the data structures that are associated with a thread and are only visible to transactions executing on that thread; the latter category includes the data structures that are associated with NUMA zones and can be possibly visible to any executing thread.

Thread-level data structures. Each thread locally stores the following data structures :

- `read_set`. The set of addresses of the objects read by the transaction currently executed by this thread.
- `write_set`. The set of pairs ⟨`addr`, `val`⟩, such that the transaction currently executed by this thread requested to write the value `val` to the object at address `addr`.
- `start_time`. An array of size equal to the total number of NUMA zones whose values can dynamically change on the execution of read operations. `start_time[i]` is the newest timestamp that is observable by the transaction currently executed by this thread for a read operation on zone i.
- `touched_zones`. An array of size equal to the total number of NUMA zones whose values can dynamically change on the execution of read operations. `touched_zones[i]` is a bitmask indicating whether the transaction currently executed by this thread has ever read or written an object that is stored in zone i.

Unless otherwise specified, each thread-level data structure is reset any time a new transaction begins.

System-level data structures. Each NUMA zone i locally stores the following data structures:

- `vcz[i]`. The vector clock of zone i. `vcz[i][i]` represents the current logical timestamp of zone i, while `vcz[i][j]` is the estimation on zone i about the current value of `vcz[j][j]` on zone j, for each $j \neq i$ (obviously `vcz[i][j]`≤`vcz[j][j]` for any j).
- `lock_table[i]`. The partition of the lock table that is stored in zone i. An entry `lock_table[i][h]` stores a pair ⟨`version`, `lock`⟩, where: *i)* version is the timestamp used by the last committed transaction that wrote to an object at an address whose hash value is equals to `h` (according to a given hash function); *ii)* lock is equal to either the id of the current thread locking an object at the address whose hash value is equal to `h`, or 0 otherwise.

5.2 Begin Operation

The begin operation of a transaction marks its starting point, it is executed anytime the transaction starts (or re-starts), and it is implemented by the `tx_begin` function, whose pseudocode is depicted in Figure 3, lines 1–2. In the pseudocode `my_zone` identifies the local NUMA zone.

At this stage the transaction copies the status of the vector clock associated with the local NUMA zone `vcz[my_zone]` to the `starting_time` array. This array defines the serialization point of the transaction. At this stage, the value of `starting_time[j]` is not definitive yet, for any zone `j` different from the local one, because that will be fixed as soon as the transaction accesses objects in zone `j` for the first time. On the contrary, `starting_time[my_zone]` is fixed, and indicates that the transaction is serialized after all the transactions that committed so far on the local zone.

5.3 Transactional Write and Read Operations

Nemo adopts a lazy versioning policy such that the result of a write operation is applied to the shared memory only during the commit phase. Therefore, a write operation of a value `val` on an object at address `addr`, which is depicted by the `tx_write` function at lines 28–29 in the pseudocode of Figure 3, simply buffers the pair `addr` and `val` in the executing thread's `write_set`. In addition, the executing thread marks that the memory of the NUMA zone containing that address as to be changed at commit time, by setting the bit `WRITE` in the bitmask associated with that zone, i.e., `touched_zones[zone_of(addr)]`. For simplicity, we suppose the existence of a function `zone_of`, which returns the identifier of a zone containing a certain address `addr`, given `addr` as input.

The behavior of a read operation in Nemo is depicted by the `tx_read` function at lines 3–27 in the pseudocode of Figure 3. When a transaction executes a read operation at an address `addr`, it first checks if `addr` is in `write_set`, and if so it returns the associated value; otherwise it checks whether it can return the current value stored in shared memory at address `addr`. In Nemo a value at a certain address `addr` can be returned if the following conditions hold: *i)* there is no concurrent write transaction that may change the value at address `addr`; *ii)* the timestamp associated with the value has been committed by a transaction that does not follow the transaction reader in the final serialization order.

If we name `zone` as the zone where `addr` is (line 5), and `h` as the hash value of `addr` (line 6), the read retrieves the `entry` associated with `h` in the `lock_table` of zone `zone`, and it checks the aforementioned conditions as follows: *i)* the timestamp stored

```
tx_begin()
1.  foreach (z in numa_zones)
2.     start_time[z] = vcz[my_zone][z];

tx_read(addr)
3.  if (write_set.exist(addr))
4.      return write_set.find(addr);
5.  zone = zone_of(addr);
6.  h = hash(addr);
7.  entry = &lock_table[zone][h];
8.  v1 = entry->version;
9.  val = *addr;
10. v2 = entry->version;
11. if (v1 != v2 || entry->lock)
12.    tx_abort();
13. if (v1 > start_time[zone])
14.    if (zone != my_zone)
15.        update_my_vector_on_read(zone,v1);
16.       if (!(touched_zones[zone] & READ))
17.          foreach (z in touched_zones)
18.             if (vcz[zone][z] > start_time[z])
19.                tx_abort();
20.          start_time[zone] = vcz[zone][zone];
21.      else
22.         tx_abort();
23.    else
24.       tx_abort();
25. touched_zones[zone] |= READ;
26. read_set.add(addr);
27. return val;

tx_write(addr, val)
28. write_set.add(addr, val);
29. touched_zones[zone_of(addr)] |= WRITE;

tx_abort()
30. foreach (w in write_set)
31.    if (w.acquired)
32.        zone = zone_of(w.addr);
33.        h = hash(w.addr);
34.        lock_table[zone][h].lock = 0; //unlock
35. tx_restart()

void  update_my_vector_on_read(zone,v)
36. if (vcz[my_zone][zone] < v)
37.     vcz[my_zone].lock();
38.     vcz[my_zone][zone] = vcz[zone][zone];
39.     vcz[my_zone].unlock();
```

Figure 3: Begin, read, write and abort operations of transactions in Nemo.

in the field `version` of `entry` remains unchanged while the read retrieves the current value at address `addr` (lines 8–11), and the `lock`'s value in `entry` is equal to 0 (line 11); *ii)* the timestamp stored in the field `version` of `entry` is either lesser than or equal to `start_time[zone]`, or if that is not the case, the read is in execution on a remote zone (lines 13–14), this is the first time the current transaction is executing a read on zone `zone` (line 16), and the current snapshot on that zone did not overwrite (either directly or transitively) any snapshot already observed so far by the current transaction on other zones (lines 17-18).

If both the conditions hold, the read can return the current value at `addr`, by also adding `addr` in `read_set` and setting the bit `READ` in the bitmask associated with `zone` (lines 25–27). In any other case, the transaction running the read aborts (lines 12, 19, 22 and 24).

In case this is the first time the current transaction is reading on that remote `zone`, and the read does not cause the abort of the transaction, then the value at `start_time[zone]` can be possibly advanced to reflect the final serialization order of the transaction with respect to the other transactions committed on `zone` (line 20). In addition, in case the read is executed on a remote zone and it finds an updated value of the remote zone's local timestamp, it copies the new value in the corresponding location of the local vector clock `vcz[my_zone]`. The update is executed by the function `update_my_vector_on_read`, and in the basic version of Nemo is executed in mutual exclusion by acquiring a lock associated with the local vector clock `vcz[my_zone]` (lines 36–39). In Section 5.5 we see how we can implement the update in a lock-free manner.

5.4 Commit and Abort Operations

Upon a commit request, Nemo executes the `tx_commit` function, whose pseudocode is depicted at lines 1–33 of Figure 4. The commit first distinguishes whether the transaction requesting the commit is read-only or not (line 1). In the case of a read-only transaction, it can commit if the values stored at addresses previously accessed by the transaction via read operations did not change since the transaction read them, and there is no concurrent write transaction that is going to modify one of them. Unlike other timestamp-based TM algorithms (e.g., TL2), in Nemo validation is required for read-only transactions due to the absence of a single centralized timestamp.

In case the transaction requesting the commit is a write transaction, the `tx_commit` behaves as follows (lines 8–33): for the entries of the lock table associated with the hash values of the addresses in the `write_set`, Nemo tries to acquire the lock by writing via a CAS primitive the transaction id as the value of the `lock` field (lines 8–16). If the procedure fails for at least one address, then the transaction aborts. Otherwise the execution proceeds and if the transaction also passes the validation (lines 17–22, executed as in the case of read-only transactions), modifications stored in the `write_set` (line 23) are applied the vector clocks associated with the modified NUMA zones is updated to reflect that a new snapshot has been created on those zones (lines 24–27).

As last step, the transaction also releases the previously acquired locks after having associated a new timestamp with the modified addresses. The new timestamp is the result of the function `update_vectors_on_commit` modifying the vector clocks. The lock-based implementation of the function basically finds a value greater than the maximum timestamp among the ones of all the entries `j` of any zone `z`'s local vector clock, where both `j` and `z` are identifiers of zones modified by the committing transaction (lines 34–45). In Section 5.5 we see how we can implement `update_vectors_on_commit` in a lock-free manner.

5.5 Lock-free Version of the Vector Clocks Updates

Locking the vector clocks for updates is costly, and can lead to a non-scalable performance in configurations with a high number of threads; thus, Nemo also supports a lock-free version of the functions updating the vector clocks.

The function `update_my_vector_on_read` updates the entry `z` that are outdated in the local vector clock `vcz[my_zone]` by advancing them to at least a certain timestamp `v`. Nemo can iteratively try to perform the change via a CAS primitive until either the CAS succeeds or another thread advances `vcz[my_zone][z]`'s value to a new value greater than or equal to `v` (lines 1–5 in Figure 5).

A lock-free version of `update_vectors_on_commit` can be implemented by exploiting the following observations: *i)* during the finalization of a commit, for each zone `z` affected by the commit, we have to first atomically increment the zone `z`'s local clock, namely `vcz[z][z]`, to guarantee that the transactions starting after the completion of the commit are able to observe the changes of the commit (lines 6–7 in Figure 5); *ii)* afterwards we have to update the local view of an affected zone `z`, i.e., `vcz[z][y]`, against any other affected zone `y`'s local clock, i.e., `vcz[y][y]` (lines 8–16 in Figure 5); *iii)* we do not need to ensure that the updates of point

```
tx_commit()
1. if (write_set.is_empty()) //read-only tx
       foreach (r in read_set) //validate read-set
2.         zone = zone_of(r.addr);
3.         h = hash(r.addr);
4.         entry = &lock_table[zone][h];
5.         if (entry->version > start_time[zone]
                  || (entry->lock && entry->lock != id))
6.             tx_abort();
7.     return;
8. foreach (w in write_set)
9.      zone = zone_of(w.addr);
10.     h = hash(w.addr);
11.     entry = &lock_table[zone][h];
12.     if (entry->lock == id) continue;
13.     if (!CAS(entry->lock, 0, id))
14.         tx_abort();
15.     else
16.         w.acquired = true;
17. foreach (r in read_set) //validate read-set
18.     zone = zone_of(r.addr);
19.     h = hash(r.addr);
20.     entry = &lock_table[zone][h];
21.     if (entry->version > start_time[zone]
              || (entry->lock && entry->lock != id))
22.         tx_abort();
23. write_set.writeback();
24. foreach (z in numa_zones)
25.     if (touched_zones[z] & WRITE)
26.         tx_zones.add(z);
27. end_ts = update_vectors_on_commit(tx_zones);
28. foreach (w in write_set)
29.     zone = zone_of(w.addr);
30.     h = hash(w.addr);
31.     entry = &lock_table[zone][h];
32.     entry->version = end_ts[zone];
33.     entry->lock = 0; //unlock

timestamp[] update_vectors_on_commit(zones)
34. max_ts = 0;
35. foreach (z in zones)
36.    vcz[z].lock();
37.    if (vcz[z][z] > max_ts)
38.          max_ts = vcz[z][z];
39. max_ts++;
40. foreach (z in zones)
41.   foreach (y in zones)
42.      vcz[z][y] = max_ts;
43.   end_ts[z] = max_ts;
44.   vcz[z].unlock();
45. return end_ts;
```

Figure 4: Commit operation of transactions in Nemo.

```
void    update_my_vector_on_read(zone,v)
1. ts = vcz[my_zone][zone];
2.     while (ts < v)
3.         old_val = CAS(vcz[my_zone][zone], ts,
                              vcz[zone][zone]);
4.         if (old_val > ts)
5.             ts = old_val;

timestamp[]    update_vectors_on_commit(zones)
6. foreach (z in zones)
7.    end_ts[z] = atomic_inc(vcz[z][z]);
8. foreach (z in zones)
9.    foreach (y in zones)
10.       if (z != y)
11.          ts = vcz[z][y];
12.          while (ts < end_ts[y])
13.             old_val = CAS(vcz[z][y],
                              ts, end_ts[y]);
14.             if (old_val > ts)
15.                ts = old_val;
16. return end_ts;
```

Figure 5: Lock-free updates of vector clocks in Nemo.

i) and point *ii)* are executed as they were in one atomic block, because the locks acquired on the addresses in `write_set` (lines 8–16 Figure 4) prevent any other concurrent read to observe at the same time both an address being updated by the commit and a partial update on the vector clocks.

6 CORRECTNESS ARGUMENTS

Due to space constraints we do not provide a formal proof on the correctness of Nemo. However we informally provide that intuition on why the histories produced by Nemo satisfy virtual world consistency. Under virtual world consistency each transaction reads from a state that is produced by a serializable history of committed write transactions, while the history restricted to the committed transactions satisfies opacity.

First of all we can easily show that each transaction in Nemo reads from a state produced by a serializable history of committed write transactions because, each transaction is only allowed to observe the state of a NUMA zone z that was already present at the time the transaction executed the first read operation on z, for any z (see lines 13–24 in Figure 3); in addition no partial updates are visible because a read operation that finds a lock acquired causes the abort of the executing transaction (see line 11 in Figure 3).

Then to prove that any history restricted to the committed transactions satisfies opacity, we show that each committed transaction appears as executed at a physical timestamp between its beginning and its commit. We briefly introduce the concepts of direct dependencies between two transactions T_i and T_k. We say that: T_k directly read-depends on T_i if T_k read a value that T_i previously wrote; T_k directly write-depends on T_i if T_k overwrote a value that T_i previously wrote; T_k directly anti-depends on T_i if T_k overwrote a value that T_i previously read; T_k real-time-depends on T_i if T_k started its execution after T_i successfully committed.

We say that a transaction T_k observed a transaction T_i if either T_k directly read-depends on T_i or T_k directly write-depends on T_i; we say that T_k did not observe a transaction T_i if T_i directly anti-depends on T_k. The reader can notice that T_k can observe T_i only if T_i is a committed write transaction in Nemo, since write operations are externalized only at commit time; in addition those are the only kinds of dependencies T_k can be aware of during its execution since read operations are invisible. We say that T_k observed T_i at timestamp $\mathcal{ST}(T_i)$, if $\mathcal{ST}(T_i)$ is a physical timestamp greater than T_i's start time and lesser than the physical timestamp at the time T_k observed T_i; otherwise we say that T_k did not observe T_i at timestamp $\mathcal{ST}(T_i)$ if $\mathcal{ST}(T_i)$ is a physical timestamp greater than or equal to the physical timestamp at the time T_i developed the anti-dependency on T_k. Finally we combine the two concepts and we say that T_k perceived T_i at timestamp $\mathcal{ST}(T_i)$, if either T_k observed T_i at $\mathcal{ST}(T_i)$ or T_k did not observe T_i at $\mathcal{ST}(T_i)$.

To prove that any execution under Nemo satisfies opacity we show that for any pair of transactions T_i and T_j, if there exists a committed transaction T_k that perceived $\mathcal{ST}(T_i) < \mathcal{ST}(T_j)$, then any other committed transaction T_w perceived $\mathcal{ST}(T_i) < \mathcal{ST}(T_j)$, for any possible choice of $\mathcal{ST}(T_i)$ and $\mathcal{ST}(T_j)$. We distinguish 5 cases depending on whether T_j directly depends on T_i or not, and on the kind of direct dependency T_i and T_j developed (if any).

T_j directly read-depends on T_i. We can choose $\mathcal{ST}(T_i)$, resp. $\mathcal{ST}(T_j)$, as the physical time when T_i, resp. T_j, releases the first lock (see the first iteration of line 33 of the `tx_commit` function in Figure 4), because both read and write operations by committed transactions are finalized on unlocked addresses (see lines 5–12 in Figure 3, and lines 8–16 of Figure 4). This is valid for both the read operation creating the read-dependency between T_i and T_j, and any perception by transactions T_k, T_w of transactions T_i and T_j. Obviously $\mathcal{ST}(T_i) < \mathcal{ST}(T_j)$ holds by construction, and because T_j can only develop the read-dependency after T_i starts to release the locks at commit time.

T_j directly write-depends on T_i. This case is analogous to the previous one since T_j can develop the write-dependency only after

T_i starts to release the locks at commit time.

T_j directly anti-depends on T_i. In this case we have to first exclude the scenario in which T_j acquires the lock on the address which the anti-dependency builds on before T_i validates that address, otherwise T_i would not commit (see lines 17–22 in Figure 4). On the other hand, in the scenario T_j starts its lock acquisition after T_i starts releasing the locks, then this case is equivalent to the previous two cases. The remaining and most interesting scenario is where T_j starts its lock acquisition concurrently with T_i's lock acquisition and validation, and possibly complete its commit phase before T_i does. However for that scenario we can choose $\mathcal{ST}(T_i)$ as the physical time when T_i releases the first lock, while $\mathcal{ST}(T_j)$ as the maximum value between the physical time when T_j releases the first lock and $\mathcal{ST}(T_i) + \varepsilon$, for any $\varepsilon > 0$. This is because any committed transactions that has to observe both T_i and T_j finalizes the observation only on unlocked addresses (see lines 5–12 in Figure 3, and lines 8–16 of Figure 4).

T_j real-time-depends on T_i. This case is equivalent to the first and second case since transaction T_j starts after transaction T_i committed, and hence if we choose $\mathcal{ST}(T_i)$, resp. $\mathcal{ST}(T_j)$, as the physical time when T_i, resp. T_j, releases the first lock, then $\mathcal{ST}(T_i) < \mathcal{ST}(T_j)$ holds for any observer, by the real-time-dependency between T_i and T_j.

T_j does not depend on T_i. In this case let us suppose that T_k perceives T_i at timestamp $\mathcal{ST}(T_i)$, resp. T_j at timestamp $\mathcal{ST}(T_j)$, while there exists T_w that perceives T_i at timestamp $\mathcal{ST}'(T_i)$, resp. T_j at timestamp $\mathcal{ST}'(T_j)$.

Let us consider that $\mathcal{ST}(T_i) < \mathcal{ST}(T_j)$ and $\mathcal{ST}'(T_i) > \mathcal{ST}'(T_j)$. This is an admissible scenario for Nemo, but if that is true, then at least one of the transactions between T_k and T_w cannot commit, by contradicting that both are committed transactions. This is because the scenario can happen only if T_k observed T_i and did not observe T_j, while T_w observed T_j and did not observe T_i, and in that case at least one between T_k and T_w aborts thanks to the validation at lines 1–6 and 17–22 in Figure 4.

7 EVALUATION

We implemented Nemo concurrency control in C++ and we integrated it into the RSTM framework [32].

We conducted a comprehensive evaluation using the following benchmarks: Bank, Linked-List, and TPC-C. *Bank* mimics a monetary application that transfers amounts of money among bank accounts. We modified Bank to be NUMA-local by partitioning accounts among NUMA zones. Inter-NUMA zone operations represent operations that work on accounts stored on different NUMA zones (e.g., a transfer from an account in the NUMA zone z to an account in the NUMA zone y).

NUMA-Linked-list is a benchmark where we partitioned a linked-list in multiple shards, and we distributed the shards among the NUMA zones. Inter-NUMA zone operations represent the ones that move a node of the linked-list from one NUMA zone to another one (e.g., a remove operation from the shard on the NUMA zone z followed by an add operation in the shard on the NUMA zone y).

TPC-C [31] is the famous on-line transaction processing (OLTP) benchmark which simulates an order-entry environment with several warehouses. TPC-C includes five transaction profiles, where three of them are write transactions and the remaining are read-only transactions. We modified TPC-C to be NUMA-local by vertically partitioning the database tables: we created a mapping from subsets of warehouses (and their related data) to NUMA zones, and such that a warehouse (and its related data) is located in one NUMA zone. We represent inter-NUMA zone operations by running a transaction originated on one zone on a warehouse maintained by a different zone.

As competitors, we considered two state-of-the-art STM protocols that rely on global shared metadata, i.e., TL2 [10] and NOrec [7]; two disjoint-access parallelism protocols, i.e., TLC [2] and Strict 2-Phase Locking (2PL) [3]; an optimized version of TL2 that was designed to alleviate the frequency of accesses to the shared global metadata, i.e., TL2 GV5 [10, 19]; and we also developed a version of NOrec enhanced with our implementation of the NUMA-aware lock of [9] (specifically C-BO-BO Lock [9]). In addition we refer to Nemo as the basic version of the algorithm, and to NemoLF as the version of Nemo having the lock-free implementation of the update functions on the vector clocks.

In this evaluation study we use the 64-core machine described in Section 2. It has four AMD Opteron 6376 Processors (2.3 GHz) and 128 GB of memory. This machine has 8 NUMA zones (2 per chip) and each NUMA zone has 16 GB of the memory. The code is compiled with GCC 4.8.2 using the O3 optimization level. We ran the experiments using Ubuntu 14.04 LTS and libnuma. All data points are the average of 5 repeated executions.

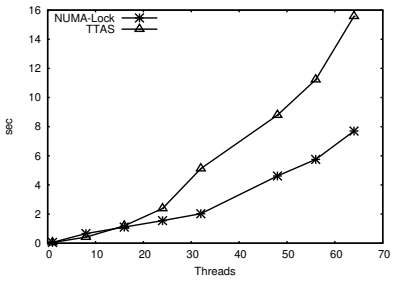

Figure 6: Comparison of a centralized global lock and a global NUMA-lock.

The NUMA-aware lock implementation is not available, so we implemented it by using the provided description in [9]. The tests of our implementation of the NUMA-lock show that it is working correctly. Figure 6 shows a comparison of a centralized lock and our implementation of the NUMA-lock. In this experiment, each thread tries to acquire the lock 100,000 times, performs some dummy work, and then releases the lock. The reported data is the average time spent by each thread to finish the task; thus lower is better. It is worth noting that threads are distributed among NUMA zones in a round-robin fashion (e.g., the configuration with 8 threads means 1 thread per NUMA zone, while 16 threads means 2 threads per NUMA zone). Thus, lock cohorting of the NUMA-lock has no benefits at 1 and 8 threads.

7.1 Bank

In this benchmark, each transaction produces 10 transfer operations accessing 20 random NUMA-local bank accounts. Each NUMA zone

has 1 million accounts (a total of 8 million accounts). In addition, 10% of the transactions are inter-NUMA zone, which means that they invoke a transfer operation involving accounts stored on two different NUMA zones.

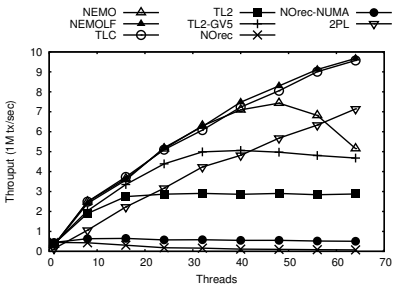

Figure 7: Throughput using Bank benchmark.

Figure 7 shows the results of the execution of the aforementioned workload by varying the total number of working threads. We notice that NemoLF provides the highest throughput, and the best scalability since it is able to increase the throughput when the scale of the system increases. On the other hand, Nemo provides comparable performance up to 40 threads, but beyond that configuration it starts to suffer due to the contention on vector clocks' locks.

TLC has performance that is very close to the one of NemoLF since the level of contention is low, and thus TLC does not suffer from a high number of unnecessary aborts. Also 2PL scales well, but the overhead of acquiring locks on both read and written objects at encounter-time is evident and therefore it shows a much lower throughput if compared to NemoLF and TLC. On the contrary, TL2-GV5 stopped scaling with at 32 threads as it is still using a centralized single timestamp, while TL2 has the same behavior starting from a size equals to 16 threads (a single socket). NOrec and NOrec with NUMA-lock (NOrec-NUMA) have very limited scalability up to 16 threads. It is clear that using a NUMA-aware lock enhances performance, but still the sequential commit phase in NOrec represents the bottleneck in such a write-dominated workload.

7.2 NUMA-Linked-list

In this benchmark, each NUMA zone has a sorted linked-list of 10000 elements, such that initially each linked-list is half-empty. Each transaction generally performs an *insert* with probability 30%, a *remove* with probability 30%, and a *contains* with probability 40% on a single linked-list. Inter-NUMA zone transactions remove an item from one linked-list and add it to another linked-list.

Each transaction traverses the linked-list from the head of the list up to the desired node (if any), and all the visited nodes are monitored by the concurrency control scheme adopted (either via insertion in the `read_set` or via acquisition of read locks). Therefore, given also the large size of the linked-lists, the transaction execution time is long, and the contention level is high.

Figure 8 shows the results of the execution of the aforementioned workload by varying the total number of application threads. With this workload, we notice that all approaches cannot scale well. This

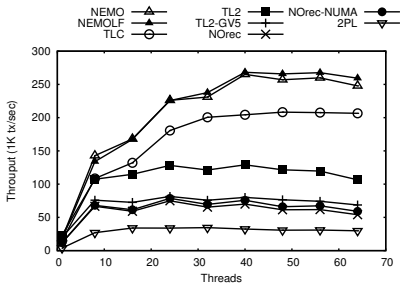

Figure 8: Throughput using NUMA-Linked-list benchmark.

is because the level of contention is high and the cost of aborting a transaction is substantial due to the long duration of transactions. Our Nemo and NemoLF approaches outperform competitors in terms of scalability and overall throughput. Even if TLC is able to sustain the throughput with a system size greater than 32, it suffers from high abort rates because of the outdated timestamps cached by the threads.

TL2 shows no scalability beyond 8 threads, and TL2-GV5 also suffers from high number of aborts since the global timestamp is not updated by every write transaction. NOrec and NOrec-NUMA show better performance since there is a 40% of read-only transactions, which can proceed concurrently; 2PL suffers significantly in this benchmark because transactions are blocked by read-locked objects and they are eventually aborted.

7.3 TPC-C

As briefly already described in the introduction of the evaluation, we partitioned the TPC-C database such that each NUMA zone stores a subset of the warehouses and their related data: specifically, each NUMA zone stores 20 warehouses. Inter-NUMA zone transactions are transactions that execute in one zone and query or update a warehouse stored in a remote zone. TPC-C transactions are more complex than the one adopted by the previous benchmarks and longer. Our configuration with 20 warehouses produces a medium level of contention. This workload represents the sweet spot for our proposal.

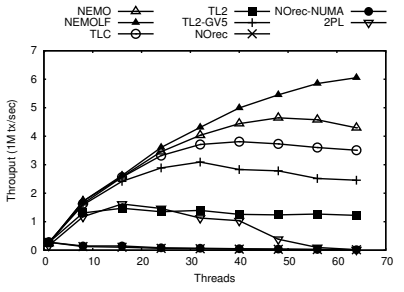

Figure 9: Throughput using TPC-C benchmark.

Figure 9 shows the results of the execution of the aforementioned workload by varying the total number of threads. Similarly to the experiments of Section 7.1, NemoLF provides good scalability and outperforms all competitors, but unlike those experiments, in

this case TLC is not able to sustain the throughput when increasing the number of threads due to a large number of unnecessary aborts caused by the pessimistic nature of its read operations. NEMO still performs better than TLC, even though it shows scalability limitation due to the locking mechanism on the vector clocks.

TL-GV5 shows the benefits of lowering the contention on centralized global metadata if compared to TL2, while both are still worse than TLC. We can also notice that 2PL suffers from moderate contention too, and the two versions of NOrec do not scale as 92% of TPC-C's default workload is composed by write transactions.

7.4 Effects of Inter-NUMA zone Transactions

In order to stress our approach in adverse scenario, in this experiment we show the effects caused by increasing the percentage of transactions accessing objects stored in different NUMA zones. Figure 10 shows the results of the Bank benchmark with the same workload profile as the one adopted in Section 7.1. We chose to adopt a system size of 48 threads so to have moderate contention in each NUMA zone but without reaching the saturation point.

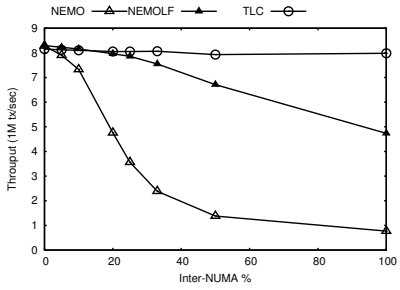

Figure 10: Throughput using Bank benchmark under different inter-NUMA-zone transactions percentage.

Clearly, our proposal was not originally designed to support a high number of non-NUMA-local transactions and we can see that from the results in Figure 10. They show that NEMO cannot scale beyond 10% because of the bottleneck introduced by the locks on the vector-clocks, while NEMOLF scales up to 23%. More in detail, by focusing on the range of 0% to 10% of Figure 10, we can notice that NEMOLF provides the best results, while TLC is slightly affected by the percentage of inter-NUMA zone transactions as it has no overhead related to updating the different NUMA zones metadata.

8 CONCLUSION

Our experiment results showed that NEMO has the best performance and scalability when the majority of the workload is scalable (i.e., it minimizes the operations involving more than one NUMA zone). In addition, we are able to achieve better performance than state-of-the art TM solutions even with moderate/high contention and complex transactional workloads.

ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367.

REFERENCES

[1] Hagit Attiya and Eshcar Hillel. 2011. Single-version STMs Can Be Multi-version Permissive. In *ICDCN '11*. 83–94.
[2] Hillel Avni and Nir Shavit. 2008. Maintaining Consistent Transactional States Without a Global Clock. In *SIROCCO '08*. 131–140.
[3] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc.
[4] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. 2013. NUMA-aware Reader-writer Locks. In *PPoPP '13*. 157–166.
[5] Kinson Chan and Cho-Li Wang. 2011. TrC-MC: Decentralized Software Transactional Memory for Multi-multicore Computers. In *ICPADS '11*. 292–299.
[6] Daniel Marcos Chapiro. 1985. *Globally-asynchronous Locally-synchronous Systems (Performance, Reliability, Digital).* Ph.D. Dissertation. Stanford, CA, USA. AAI8506166.
[7] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP '10*. 67–78.
[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *SOSP '13*. 33–48.
[9] David Dice, Virendra J. Marathe, and Nir Shavit. 2012. Lock Cohorting: A General Technique for Designing NUMA Locks. In *PPoPP '12*. 247–256.
[10] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *DISC '06*. 194–208.
[11] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. In *PLDI '09*.
[12] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. 2007. SNZI: Scalable NonZero Indicators. In *PODC '07*. 13–22.
[13] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *PPoPP '08*. 237–246.
[14] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. 2007. Database Servers on Chip Multiprocessors: Limitations and Opportunities.. In *CIDR '07*.
[15] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC '03*. 92–101.
[16] Damien Imbs and Michel Raynal. 2012. Virtual World Consistency: A Condition for STM Systems (with a Versatile Protocol with Invisible Read Operations). *Theor. Comput. Sci.* 444 (July 2012), 113–127.
[17] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *PODC '94*. 151–160.
[18] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. 2013. Experimental Evaluation of NUMA Effects on Database Management Systems. In *BTW '13*. 185–204.
[19] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a Scalable Software Transactional Memory. In *TRANSACT '09*.
[20] Kai Lu, Ruibo Wang, and Xicheng Lu. 2010. Brief announcement: NUMA-aware Transactional Memory.. In *PODC '10*. 69–70.
[21] Nakul Manchanda and Karan Anand. 2010. Non-Uniform Memory Access (NUMA). *New York University* (2010).
[22] Patrick Marlier, Anita Sobe, and Pierre Sutra. 2014. A Locality-Aware Software Transactional Memory. In *WTM '14*.
[23] Mohamed Mohamedin, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran. 2016. On designing NUMA-aware concurrency control for scalable transactional memory. In *ACM SIGPLAN PPoPP*. ACM, 45:1–45:2.
[24] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. 2015. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. In *PODC '15*. 217–226.
[25] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tözün, and Anastasia Ailamaki. 2012. OLTP on Hardware Islands. *PVLDB* 5, 11 (2012), 1447–1458.
[26] Torvald Riegel, Christof Fetzer, and Pascal Felber. 2007. Time-based Transactional Memory with Scalable Time Bases. In *SPAA '07*.
[27] Wenjia Ruan, Yujie Liu, and Michael Spear. 2013. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013), 40:1–40:21.
[28] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. 2011. Database Engines on Multicores, Why Parallelize when You Can Distribute?. In *EuroSys '11*. 17–30.
[29] M.F. Spear, V.J. Marathe, L. Dalessandro, and M.L. Scott. 2007. Privatization Techniques for Software Transactional Memory. In *PODC '07*. 338–339.
[30] Michael F. Spear, Maged M. Michael, and Christoph von Praun. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *SPAA '08*. 275–284.
[31] TPC Council. 2010. TPC-C Benchmark, Revision 5.11. (Feb. 2010).
[32] University of Rochester. 2006. Rochester Software Transactional Memory. http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml, http://code.google.com/p/rstm. (2006).